目　录

安装

gRPC简介

- gRPC由google开发，是一款语言中立、平台中立、开源的远程过程调用系统
- gRPC客户端和服务端可以在多种环境中运行和交互，例如用java写一个服务端，可以用go语言写客户端调用

gRPC与Protobuf介绍

- 微服务架构中，由于每个服务对应的代码库是独立运行的，无法直接调用，彼此间的通信就是个大问题
- gRPC可以实现微服务，将大的项目拆分为多个小且独立的业务模块，也就是服务，各服务间使用高效的protobuf协议进行RPC调用，gRPC默认使用protocol buffers，这是google开源的一套成熟的结构数据序列化机制（当然也可以使用其他数据格式如JSON）
- 可以用proto files创建gRPC服务，用message类型来定义方法参数和返回类型

安装gRPC和Protobuf

- go get github.com/golang/protobuf/proto
- go get google.golang.org/grpc（无法使用，用如下命令代替）
    - git clone https://github.com/grpc/grpc-go.git $GOPATH/src/google.golang.org/grpc
    - git clone https://github.com/golang/net.git $GOPATH/src/golang.org/x/net
    - git clone https://github.com/golang/text.git $GOPATH/src/golang.org/x/text
    - go get -u github.com/golang/protobuf/{proto,protoc-gen-go}
    - git clone https://github.com/google/go-genproto.git $GOPATH/src/google.golang.org/genproto
    - cd $GOPATH/src/
    - go install google.golang.org/grpc
- go get github.com/golang/protobuf/protoc-gen-go
- 上面安装好后，会在GOPATH/bin下生成protoc-gen-go.exe

- 但还需要一个protoc.exe，windows平台编译受限，很难自己手动编译，直接去网站下载一个，地址：https://github.com/protocolbuffers/protobuf/releases/tag/v3.9.0 ，同样放在GOPATH/bin下

注意：这里面好多都是需要vpn才能下载好的！分享一个下载好的
https://pan.baidu.com/s/1T8eJkHib2uPL3gNMCRdZmQ 提取码 s42z

gRPC简介

这里使用一个测试文件对照说明常用结构的protobuf到golang的转换。只说明关键部分代码，详细内容请查看完整文件。示例文件在 `proto/test` 目录下。

Package

在proto文件中使用 `package` 关键字声明包名，默认转换成go中的包名与此一致，如果需要指定不一样的包名，可以使用 `go_package` 选项：

```
package test;
option go_package="test";
```

Message

proto中的 `message` 对应go中的 `struct` ，全部使用驼峰命名规则。嵌套定义的 `message` ， `enum` 转换为go之后，名称变为 `Parent_Child` 结构。

示例proto：

```
// Test 测试
message Test {
    int32 age = 1;
    int64 count = 2;
    double money = 3;
    float score = 4;
    string name = 5;
    bool fat = 6;
    bytes char = 7;
    // Status 枚举状态
    enum Status {
        OK = 0;
        FAIL = 1;
    }
    Status status = 8;
    // Child 子结构
    message Child {
        string sex = 1;
    }
    Child child = 9;
    map<string, string> dict = 10;
}
```

转换结果：

```
// Status 枚举状态
type Test_Status int32

const (
    Test_OK   Test_Status = 0
    Test_FAIL Test_Status = 1
)

// Test 测试
type Test struct {
    Age    int32        `protobuf:"varint,1,opt,name=age" json:"age,omitempty"`
    Count  int64        `protobuf:"varint,2,opt,name=count" json:"count,omitempt
y"`
    Money  float64      `protobuf:"fixed64,3,opt,name=money" json:"money,omitempt
y"`
    Score  float32      `protobuf:"fixed32,4,opt,name=score" json:"score,omitempt
y"`
    Name   string       `protobuf:"bytes,5,opt,name=name" json:"name,omitempty"`
    Fat    bool         `protobuf:"varint,6,opt,name=fat" json:"fat,omitempty"`
    Char   []byte       `protobuf:"bytes,7,opt,name=char,proto3" json:"char,omite
mpty"`
    Status Test_Status `protobuf:"varint,8,opt,name=status,enum=test.Test_Statu
s" json:"status,omitempty"`
    Child  *Test_Child `protobuf:"bytes,9,opt,name=child" json:"child,omitempty"
`
    Dict   map[string]string `protobuf:"bytes,10,rep,name=dict" json:"dict,omite
mpty" protobuf_key:"bytes,1,opt,name=key" protobuf_val:"bytes,2,opt,name=value"`
}

// Child 子结构
type Test_Child struct {
    Sex string `protobuf:"bytes,1,opt,name=sex" json:"sex,omitempty"`
}
```

除了会生成对应的结构外，还会有些工具方法，如字段的getter:

```
func (m *Test) GetAge() int32 {
    if m != nil {
        return m.Age
    }
    return 0
}
```

枚举类型会生成对应名称的常量，同时会有两个map方便使用：

```
var Test_Status_name = map[int32]string{
    0: "OK",
    1: "FAIL",
}
var Test_Status_value = map[string]int32{
    "OK":   0,
    "FAIL": 1,
}
```

Service

定义一个简单的Service，`TestService`有一个方法`Test`，接收一个`Request`参数，返回`Response`：

```
// TestService 测试服务
service TestService {
    // Test 测试方法
    rpc Test(Request) returns (Response) {};
}

// Request 请求结构
message Request {
    string name = 1;
}

// Response 响应结构
message Response {
    string message = 1;
}
```

转换结果：

```
// 客户端接口
type TestServiceClient interface {
    // Test 测试方法
    Test(ctx context.Context, in *Request, opts ...grpc.CallOption) (*Response,
 error)
}

// 服务端接口
type TestServiceServer interface {
    // Test 测试方法
```

```
    Test(context.Context, *Request) (*Response, error)
}
```

生成的go代码中包含该Service定义的接口，客户端接口已经自动实现了，直接供客户端使用者调用，服务端接口需要由服务提供方实现。

Protobuf⇢Go转换

这里使用一个测试文件对照说明常用结构的protobuf到golang的转换。只说明关键部分代码，详细内容请查看完整文件。示例文件在 `proto/test` 目录下。

Package

在proto文件中使用 `package` 关键字声明包名，默认转换成go中的包名与此一致，如果需要指定不一样的包名，可以使用 `go_package` 选项：

```
package test;
option go_package="test";
```

Message

proto中的 `message` 对应go中的 `struct` ，全部使用驼峰命名规则。嵌套定义的 `message` ， `enum` 转换为go之后，名称变为 `Parent_Child` 结构。

示例proto：

```
// Test 测试
message Test {
    int32 age = 1;
    int64 count = 2;
    double money = 3;
    float score = 4;
    string name = 5;
    bool fat = 6;
    bytes char = 7;
    // Status 枚举状态
    enum Status {
        OK = 0;
        FAIL = 1;
    }
    Status status = 8;
    // Child 子结构
    message Child {
        string sex = 1;
    }
    Child child = 9;
    map<string, string> dict = 10;
}
```

转换结果：

```
// Status 枚举状态
type Test_Status int32

const (
    Test_OK   Test_Status = 0
    Test_FAIL Test_Status = 1
)

// Test 测试
type Test struct {
    Age    int32        `protobuf:"varint,1,opt,name=age" json:"age,omitempty"`
    Count  int64        `protobuf:"varint,2,opt,name=count" json:"count,omitempt
y"`
    Money  float64      `protobuf:"fixed64,3,opt,name=money" json:"money,omitempt
y"`
    Score  float32      `protobuf:"fixed32,4,opt,name=score" json:"score,omitempt
y"`
    Name   string       `protobuf:"bytes,5,opt,name=name" json:"name,omitempty"`
    Fat    bool         `protobuf:"varint,6,opt,name=fat" json:"fat,omitempty"`
    Char   []byte       `protobuf:"bytes,7,opt,name=char,proto3" json:"char,omite
mpty"`
    Status Test_Status `protobuf:"varint,8,opt,name=status,enum=test.Test_Statu
s" json:"status,omitempty"`
    Child  *Test_Child `protobuf:"bytes,9,opt,name=child" json:"child,omitempty"
`
    Dict   map[string]string `protobuf:"bytes,10,rep,name=dict" json:"dict,omite
mpty" protobuf_key:"bytes,1,opt,name=key" protobuf_val:"bytes,2,opt,name=value"`
}

// Child 子结构
type Test_Child struct {
    Sex string `protobuf:"bytes,1,opt,name=sex" json:"sex,omitempty"`
}
```

除了会生成对应的结构外，还会有些工具方法，如字段的getter:

```
func (m *Test) GetAge() int32 {
    if m != nil {
        return m.Age
    }
    return 0
}
```

枚举类型会生成对应名称的常量，同时会有两个map方便使用：

```
var Test_Status_name = map[int32]string{
    0: "OK",
    1: "FAIL",
}
var Test_Status_value = map[string]int32{
    "OK":   0,
    "FAIL": 1,
}
```

Service

定义一个简单的Service，`TestService`有一个方法`Test`，接收一个`Request`参数，返回`Response`：

```
// TestService 测试服务
service TestService {
    // Test 测试方法
    rpc Test(Request) returns (Response) {};
}

// Request 请求结构
message Request {
    string name = 1;
}

// Response 响应结构
message Response {
    string message = 1;
}
```

转换结果：

```
// 客户端接口
type TestServiceClient interface {
    // Test 测试方法
    Test(ctx context.Context, in *Request, opts ...grpc.CallOption) (*Response,
 error)
}

// 服务端接口
type TestServiceServer interface {
    // Test 测试方法
```

```
    Test(context.Context, *Request) (*Response, error)
}
```

生成的go代码中包含该Service定义的接口，客户端接口已经自动实现了，直接供客户端使用者调用，服务端接口需要由服务提供方实现。

Protobuf语法

按照惯例，这里也从一个Hello项目开始，本项目定义了一个Hello Service，客户端发送包含字符串名字的请求，服务端返回Hello消息。

流程：

1. 编写`.proto`描述文件
2. 编译生成`.pb.go`文件
3. 服务端实现约定的接口并提供服务
4. 客户端按照约定调用`.pb.go`文件中的方法请求服务

项目结构：

```
|—— hello/
    |—— client/
        |—— main.go    // 客户端
    |—— server/
        |—— main.go    // 服务端
|—— proto/
    |—— hello/
        |—— hello.proto    // proto描述文件
        |—— hello.pb.go    // proto编译后文件
```

Step1：编写描述文件：hello.proto

```
syntax = "proto3"; // 指定proto版本
package hello;     // 指定默认包名

// 指定golang包名
option go_package = "hello";

// 定义Hello服务
service Hello {
    // 定义SayHello方法
    rpc SayHello(HelloRequest) returns (HelloResponse) {}
}

// HelloRequest 请求结构
message HelloRequest {
    string name = 1;
}

// HelloResponse 响应结构
```

```
message HelloResponse {
    string message = 1;
}
```

`hello.proto` 文件中定义了一个Hello Service，该服务包含一个 `SayHello` 方法，同时声明了 `HelloRequest` 和 `HelloResponse` 消息结构用于请求和响应。客户端使用 `HelloRequest` 参数调用 `SayHello` 方法请求服务端，服务端响应 `HelloResponse` 消息。一个最简单的服务就定义好了。

Step2：编译生成 `.pb.go` 文件

```
$ cd proto/hello

# 编译hello.proto
$ protoc -I . --go_out=plugins=grpc:. ./hello.proto
```

在当前目录内生成的 `hello.pb.go` 文件，按照 `.proto` 文件中的说明，包含服务端接口 `HelloServer` 描述，客户端接口及实现 `HelloClient` ，及 `HelloRequest` 、 `HelloResponse` 结构体。

> 注意：不要手动编辑该文件

Step3：实现服务端接口 `server/main.go`

```
package main

import (
    "fmt"
    "net"

    pb "github.com/jergoo/go-grpc-example/proto/hello" // 引入编译生成的包
    "golang.org/x/net/context"
    "google.golang.org/grpc"
    "google.golang.org/grpc/grpclog"
)

const (
    // Address gRPC服务地址
    Address = "127.0.0.1:50052"
)

// 定义helloService并实现约定的接口
type helloService struct{}
```

```
// HelloService Hello服务
var HelloService = helloService{}

// SayHello 实现Hello服务接口
func (h helloService) SayHello(ctx context.Context, in *pb.HelloRequest) (*pb.He
lloResponse, error) {
    resp := new(pb.HelloResponse)
    resp.Message = fmt.Sprintf("Hello %s.", in.Name)

    return resp, nil
}

func main() {
    listen, err := net.Listen("tcp", Address)
    if err != nil {
        grpclog.Fatalf("Failed to listen: %v", err)
    }

    // 实例化grpc Server
    s := grpc.NewServer()

    // 注册HelloService
    pb.RegisterHelloServer(s, HelloService)

    grpclog.Println("Listen on " + Address)
    s.Serve(listen)
}
```

服务端引入编译后的 `proto` 包，定义一个空结构用于实现约定的接口，接口描述可以查看 `hello.pb.go` 文件中的 `HelloServer` 接口描述。实例化grpc Server并注册HelloService，开始提供服务。

运行：

```
$ go run main.go
Listen on 127.0.0.1:50052  //服务端已开启并监听50052端口
```

Step4：实现客户端调用 `client/main.go`

```
package main

import (
    pb "github.com/jergoo/go-grpc-example/proto/hello" // 引入proto包
    "golang.org/x/net/context"
    "google.golang.org/grpc"
```

```
    "google.golang.org/grpc/grpclog"
)

const (
    // Address gRPC服务地址
    Address = "127.0.0.1:50052"
)

func main() {
    // 连接
    conn, err := grpc.Dial(Address, grpc.WithInsecure())
    if err != nil {
        grpclog.Fatalln(err)
    }
    defer conn.Close()

    // 初始化客户端
    c := pb.NewHelloClient(conn)

    // 调用方法
    req := &pb.HelloRequest{Name: "gRPC"}
    res, err := c.SayHello(context.Background(), req)

    if err != nil {
        grpclog.Fatalln(err)
    }

    grpclog.Println(res.Message)
}
```

客户端初始化连接后直接调用 `hello.pb.go` 中实现的 `SayHello` 方法，即可向服务端发起请求，使用姿势就像调用本地方法一样。

运行：

```
$ go run main.go
Hello gRPC.    // 接收到服务端响应
```

如果你收到了”Hello gRPC”的回复，恭喜你已经会使用github.com/jergoo/go-grpc-example/proto/hello了。

> 建议到这里仔细看一看hello.pb.go文件中的内容，对比hello.proto文件，理解protobuf中的定义转换为golang后的结构。

小案例

OpenSSL官网

官方下载地址： https://www.openssl.org/source/

Windows安装方法

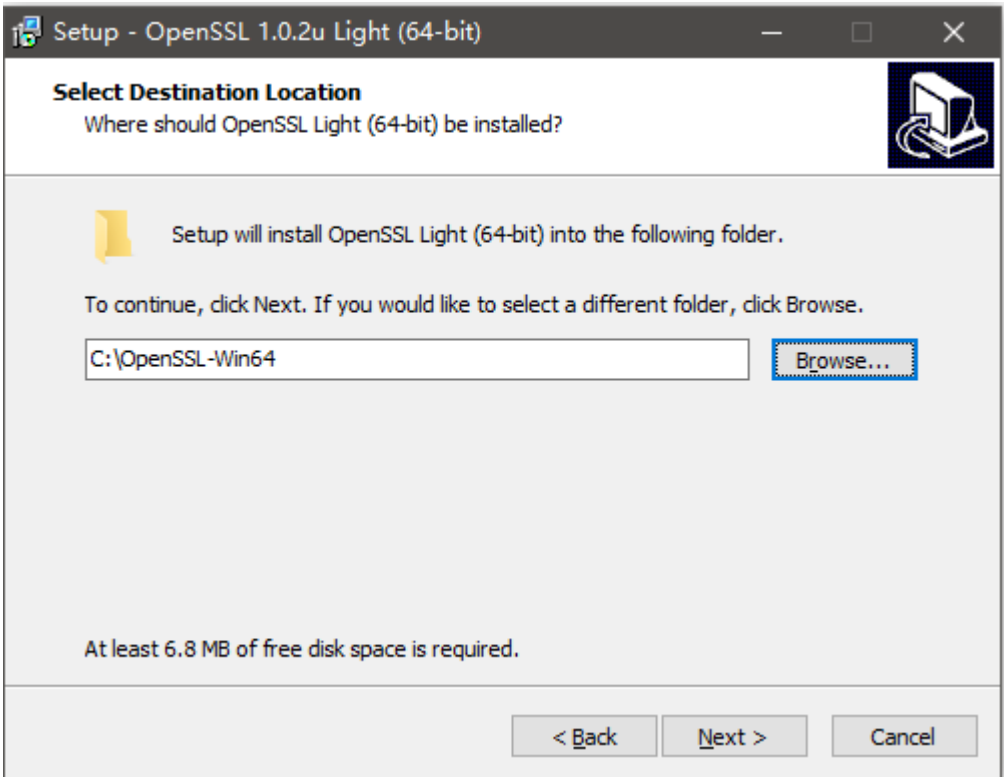
OpenSSL官网没有提供windows版本的安装包，可以选择其他开源平台提供的工具。例如

http://slproweb.com/products/Win32OpenSSL.html

以该工具为例，安装步骤和使用方法如下：

进入下载页面选择下载的版本

选择安装的位置



剩下的都是下一步

设置环境变量，例如工具安装在C:\OpenSSL-Win64，则将C:\OpenSSL-Win64\bin；复制到Path中

打开命令行程序cmd（以管理员身份运行），运行以下命令:

```
利用  openssl  生成公钥私钥
生成公钥:  openssl genrsa -out rsa_private_key.pem 1024
生成私钥: openssl rsa -in rsa_private_key.pem -pubout -out rsa_public_key.pem
```

OpenSSL安装

OpenSSL官网

官方下载地址： https://www.openssl.org/source/

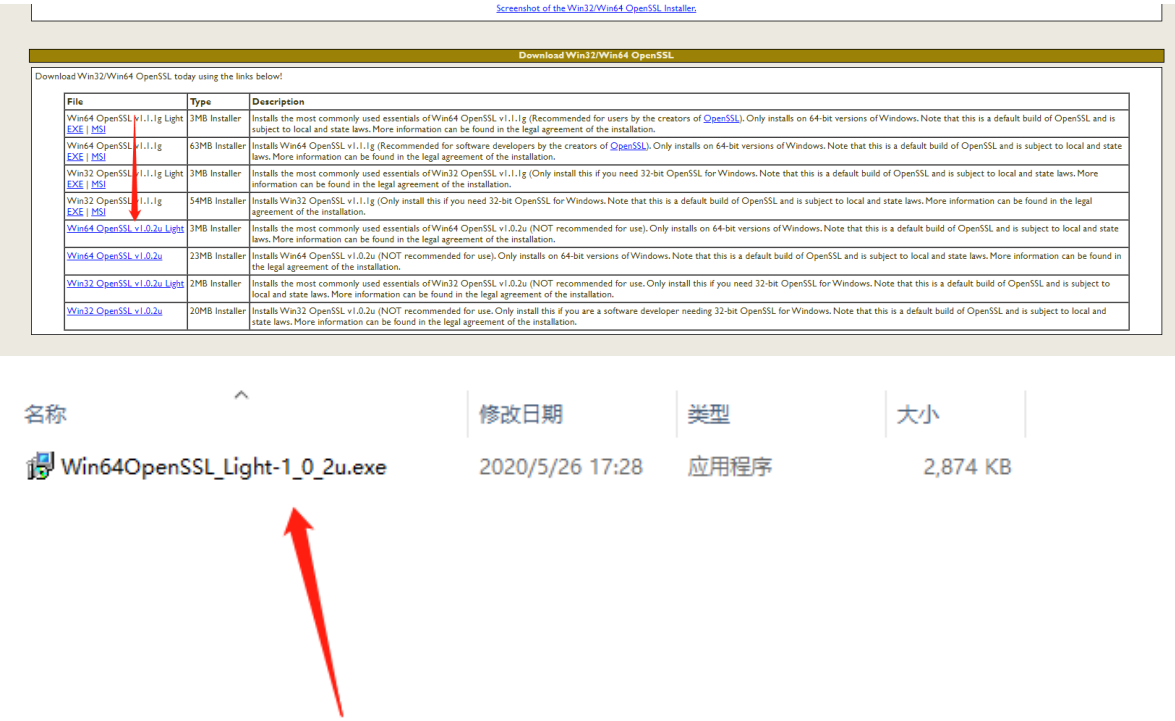
Windows安装方法

OpenSSL官网没有提供windows版本的安装包，可以选择其他开源平台提供的工具。例如

http://slproweb.com/products/Win32OpenSSL.html

以该工具为例，安装步骤和使用方法如下：

进入下载页面选择下载的版本



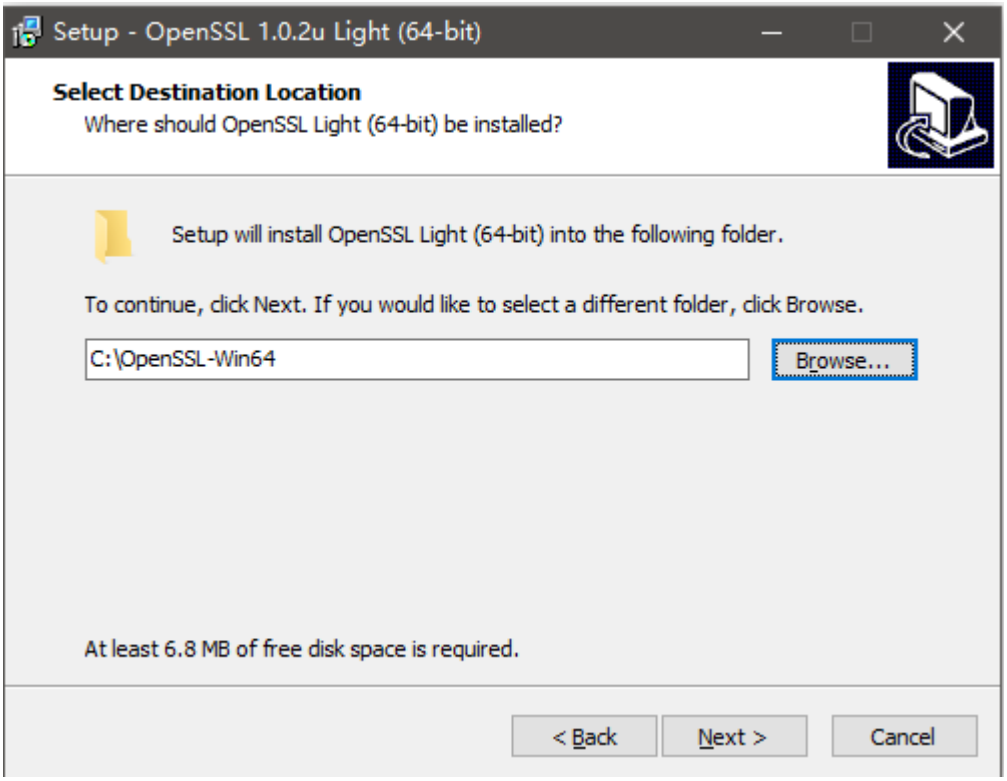
Screenshot of the Win32/Win64 OpenSSL Installer.

Download Win32/Win64 OpenSSL

Download Win32/Win64 OpenSSL today using the links below!

File	Type	Description
Win64 OpenSSL v1.1.1g Light EXE | MSI	3MB Installer	Installs the most commonly used essentials of Win64 OpenSSL v1.1.1g (Recommended for users by the creators of OpenSSL). Only installs on 64-bit versions of Windows. Note that this is a default build of OpenSSL and is subject to local and state laws. More information can be found in the legal agreement of the installation.
Win64 OpenSSL v1.1.1g EXE | MSI	63MB Installer	Installs Win64 OpenSSL v1.1.1g (Recommended for software developers by the creators of OpenSSL). Only installs on 64-bit versions of Windows. Note that this is a default build of OpenSSL and is subject to local and state laws. More information can be found in the legal agreement of the installation.
Win32 OpenSSL v1.1.1g Light EXE | MSI	3MB Installer	Installs the most commonly used essentials of Win32 OpenSSL v1.1.1g (Only install this if you need 32-bit OpenSSL for Windows. Note that this is a default build of OpenSSL and is subject to local and state laws. More information can be found in the legal agreement of the installation.
Win32 OpenSSL v1.1.1g EXE | MSI	54MB Installer	Installs Win32 OpenSSL v1.1.1g (Only install this if you need 32-bit OpenSSL for Windows. Note that this is a default build of OpenSSL and is subject to local and state laws. More information can be found in the legal agreement of the installation.
Win64 OpenSSL v1.0.2u Light	3MB Installer	Installs the most commonly used essentials of Win64 OpenSSL v1.0.2u (NOT recommended for use). Only installs on 64-bit versions of Windows. Note that this is a default build of OpenSSL and is subject to local and state laws. More information can be found in the legal agreement of the installation.
Win64 OpenSSL v1.0.2u	23MB Installer	Installs Win64 OpenSSL v1.0.2u (NOT recommended for use). Only installs on 64-bit versions of Windows. Note that this is a default build of OpenSSL and is subject to local and state laws. More information can be found in the legal agreement of the installation.
Win32 OpenSSL v1.0.2u Light	2MB Installer	Installs the most commonly used essentials of Win32 OpenSSL v1.0.2u (NOT recommended for use. Only install this if you need 32-bit OpenSSL for Windows. Note that this is a default build of OpenSSL and is subject to local and state laws. More information can be found in the legal agreement of the installation.
Win32 OpenSSL v1.0.2u	20MB Installer	Installs Win32 OpenSSL v1.0.2u (NOT recommended for use. Only install this if you are a software developer needing 32-bit OpenSSL for Windows. Note that this is a default build of OpenSSL and is subject to local and state laws. More information can be found in the legal agreement of the installation.

选择安装的位置



剩下的都是下一步

设置环境变量，例如工具安装在C:\OpenSSL-Win64，则将C:\OpenSSL-Win64\bin；复制到Path中

打开命令行程序cmd（以管理员身份运行），运行以下命令:

```
利用  openssl  生成公钥私钥
生成公钥:  openssl genrsa -out rsa_private_key.pem 1024
生成私钥: openssl rsa -in rsa_private_key.pem -pubout -out rsa_public_key.pem
```

认证

grpc服务端和客户端都提供了interceptor功能，功能类似middleware，很适合在这里处理验证、日志等流程。

在自定义Token认证的示例中，认证信息是由每个服务中的方法处理并认证的，如果有大量的接口方法，这种姿势就太不优雅了，每个接口实现都要先处理认证信息。这个时候interceptor就可以用来解决了这个问题，在请求被转到具体接口之前处理认证信息，一处认证，到处无忧。
在客户端，我们增加一个请求日志，记录请求相关的参数和耗时等等。修改hello_token项目实现：

目录结构

```
|—— hello_interceptor/
    |—— client/
        |—— main.go    // 客户端
    |—— server/
        |—— main.go    // 服务端
|—— keys/              // 证书目录
    |—— server.key
    |—— server.pem
|—— proto/
    |—— hello/
        |—— hello.proto    // proto描述文件
        |—— hello.pb.go    // proto编译后文件
```

示例代码

Step 1. 服务端interceptor：

> hello_interceptor/server/main.go

```
package main

import (
    "fmt"
    "net"

    pb "github.com/jergoo/go-grpc-example/proto/hello"

    "golang.org/x/net/context"
```

```
    "google.golang.org/grpc"
    "google.golang.org/grpc/codes"        // grpc 响应状态码
    "google.golang.org/grpc/credentials" // grpc认证包
    "google.golang.org/grpc/grpclog"
    "google.golang.org/grpc/metadata" // grpc metadata包
)

const (
    // Address gRPC服务地址
    Address = "127.0.0.1:50052"
)

// 定义helloService并实现约定的接口
type helloService struct{}

// HelloService Hello服务
var HelloService = helloService{}

// SayHello 实现Hello服务接口
func (h helloService) SayHello(ctx context.Context, in *pb.HelloRequest) (*pb.He
lloResponse, error) {
    resp := new(pb.HelloResponse)
    resp.Message = fmt.Sprintf("Hello %s.", in.Name)

    return resp, nil
}

func main() {
    listen, err := net.Listen("tcp", Address)
    if err != nil {
        grpclog.Fatalf("Failed to listen: %v", err)
    }

    var opts []grpc.ServerOption

    // TLS认证
    creds, err := credentials.NewServerTLSFromFile("../../keys/server.pem",
"../../keys/server.key")
    if err != nil {
        grpclog.Fatalf("Failed to generate credentials %v", err)
    }

    opts = append(opts, grpc.Creds(creds))

    // 注册interceptor
    opts = append(opts, grpc.UnaryInterceptor(interceptor))
```

```
    // 实例化grpc Server
    s := grpc.NewServer(opts...)

    // 注册HelloService
    pb.RegisterHelloServer(s, HelloService)

    grpclog.Println("Listen on " + Address + " with TLS + Token + Interceptor")

    s.Serve(listen)
}

// auth 验证Token
func auth(ctx context.Context) error {
    md, ok := metadata.FromContext(ctx)
    if !ok {
        return grpc.Errorf(codes.Unauthenticated, "无Token认证信息")
    }

    var (
        appid  string
        appkey string
    )

    if val, ok := md["appid"]; ok {
        appid = val[0]
    }

    if val, ok := md["appkey"]; ok {
        appkey = val[0]
    }

    if appid != "101010" || appkey != "i am key" {
        return grpc.Errorf(codes.Unauthenticated, "Token认证信息无效: appid=%s,
 appkey=%s", appid, appkey)
    }

    return nil
}

// interceptor 拦截器
func interceptor(ctx context.Context, req interface{}, info *grpc.UnaryServerInf
o, handler grpc.UnaryHandler) (interface{}, error) {
    err := auth(ctx)
    if err != nil {
        return nil, err
```

```
    }
    // 继续处理请求
    return handler(ctx, req)
}
```

Step 2. 实现客户端interceptor：

> hello_interceproor/client/main.go

```
package main

import (
    "time"

    pb "github.com/jergoo/go-grpc-example/proto/hello" // 引入proto包

    "golang.org/x/net/context"
    "google.golang.org/grpc"
    "google.golang.org/grpc/credentials" // 引入grpc认证包
    "google.golang.org/grpc/grpclog"
)

const (
    // Address gRPC服务地址
    Address = "127.0.0.1:50052"

    // OpenTLS 是否开启TLS认证
    OpenTLS = true
)

// customCredential 自定义认证
type customCredential struct{}

// GetRequestMetadata 实现自定义认证接口
func (c customCredential) GetRequestMetadata(ctx context.Context, uri ...string)
(map[string]string, error) {
    return map[string]string{
        "appid":  "101010",
        "appkey": "i am key",
    }, nil
}

// RequireTransportSecurity 自定义认证是否开启TLS
func (c customCredential) RequireTransportSecurity() bool {
    return OpenTLS
```

```
}

func main() {
    var err error
    var opts []grpc.DialOption

    if OpenTLS {
        // TLS连接
        creds, err := credentials.NewClientTLSFromFile("../../keys/server.pem",
"server name")
        if err != nil {
            grpclog.Fatalf("Failed to create TLS credentials %v", err)
        }
        opts = append(opts, grpc.WithTransportCredentials(creds))
    } else {
        opts = append(opts, grpc.WithInsecure())
    }

    // 指定自定义认证
    opts = append(opts, grpc.WithPerRPCCredentials(new(customCredential)))
    // 指定客户端interceptor
    opts = append(opts, grpc.WithUnaryInterceptor(interceptor))

    conn, err := grpc.Dial(Address, opts...)
    if err != nil {
        grpclog.Fatalln(err)
    }
    defer conn.Close()

    // 初始化客户端
    c := pb.NewHelloClient(conn)

    // 调用方法
    req := &pb.HelloRequest{Name: "gRPC"}
    res, err := c.SayHello(context.Background(), req)
    if err != nil {
        grpclog.Fatalln(err)
    }

    grpclog.Println(res.Message)
}

// interceptor 客户端拦截器
func interceptor(ctx context.Context, method string, req, reply interface{}, cc
 *grpc.ClientConn, invoker grpc.UnaryInvoker, opts ...grpc.CallOption) error {
    start := time.Now()
```

```
    err := invoker(ctx, method, req, reply, cc, opts...)
    grpclog.Printf("method=%s req=%v rep=%v duration=%s error=%v\n", method, req, reply, time.Since(start), err)
    return err
}
```

运行结果

```
$ cd hello_inteceptor/server && go run main.go
Listen on 127.0.0.1:50052 with TLS + Token + Interceptor
```

```
$ cd hello_inteceptor/client && go run main.go
method=/hello.Hello/SayHello req=name:"gRPC"  rep=message:"Hello gRPC."  duration=33.879699ms error=<nil>

Hello gRPC.
```

项目推荐： go-grpc-middleware

这个项目对interceptor进行了封装，支持多个拦截器的链式组装，对于需要多种处理的地方使用起来会更方便些。

拦截器

grpc内置了客户端和服务端的请求追踪，基于 `golang.org/x/net/trace` 包实现，默认是开启状态，可以查看事件和请求日志，对于基本的请求状态查看调试也是很有帮助的，客户端与服务端基本一致，这里以服务端开启trace server为例，修改hello项目服务端代码：

目录结构

```
|—— hello_trace/
    |—— client/
        |—— main.go    // 客户端
    |—— server/
        |—— main.go    // 服务端
|—— proto/
    |—— hello/
        |—— hello.proto    // proto描述文件
        |—— hello.pb.go    // proto编译后文件
```

示例代码

```
package main

import (
    "fmt"
    "net"
    "net/http"

    pb "github.com/jergoo/go-grpc-example/proto/hello" // 引入编译生成的包

    "golang.org/x/net/context"
    "golang.org/x/net/trace"
    "google.golang.org/grpc"
    "google.golang.org/grpc/grpclog"
)

const (
    // Address gRPC服务地址
    Address = "127.0.0.1:50052"
)

// 定义helloService并实现约定的接口
```

```
type helloService struct{}

// HelloService Hello服务
var HelloService = helloService{}

// SayHello 实现Hello服务接口
func (h helloService) SayHello(ctx context.Context, in *pb.HelloRequest) (*pb.He
lloResponse, error) {
    resp := new(pb.HelloResponse)
    resp.Message = fmt.Sprintf("Hello %s.", in.Name)

    return resp, nil
}

func main() {
    listen, err := net.Listen("tcp", Address)
    if err != nil {
        grpclog.Fatalf("failed to listen: %v", err)
    }

    // 实例化grpc Server
    s := grpc.NewServer()

    // 注册HelloService
    pb.RegisterHelloServer(s, HelloService)

    // 开启trace
    go startTrace()

    grpclog.Println("Listen on " + Address)
    s.Serve(listen)
}

func startTrace() {
    trace.AuthRequest = func(req *http.Request) (any, sensitive bool) {
        return true, true
    }
    go http.ListenAndServe(":50051", nil)
    grpclog.Println("Trace listen on 50051")
}
```

这里我们开启一个http服务监听50051端口，用来查看grpc请求的trace信息

运行：

```
$ go run main.go

Listen on 127.0.0.1:50052
Trace listen on 50051

# 进入client目录执行一次客户端请求
```

服务端事件查看

访问：localhost:50051/debug/events，结果如图：

可以看到服务端注册的服务和服务正常启动的事件信息。

请求日志信息查看

访问：localhost:50051/debug/requests，结果如图：

这里可以显示最近的请求状态，包括请求的服务、参数、耗时、响应，对于简单的状态查看还是很方便的，默认值显示最近10条记录。

内置Trace

grpc内置了客户端和服务端的请求追踪，基于 `golang.org/x/net/trace` 包实现，默认是开启状态，可以查看事件和请求日志，对于基本的请求状态查看调试也是很有帮助的，客户端与服务端基本一致，这里以服务端开启trace server为例，修改hello项目服务端代码：

目录结构

```
|—— hello_trace/
    |—— client/
        |—— main.go    // 客户端
    |—— server/
        |—— main.go    // 服务端
|—— proto/
    |—— hello/
        |—— hello.proto    // proto描述文件
        |—— hello.pb.go    // proto编译后文件
```

示例代码

```
package main

import (
    "fmt"
    "net"
    "net/http"

    pb "github.com/jergoo/go-grpc-example/proto/hello" // 引入编译生成的包

    "golang.org/x/net/context"
    "golang.org/x/net/trace"
    "google.golang.org/grpc"
    "google.golang.org/grpc/grpclog"
)

const (
    // Address gRPC服务地址
    Address = "127.0.0.1:50052"
)

// 定义helloService并实现约定的接口
```

```
type helloService struct{}

// HelloService Hello服务
var HelloService = helloService{}

// SayHello 实现Hello服务接口
func (h helloService) SayHello(ctx context.Context, in *pb.HelloRequest) (*pb.He
lloResponse, error) {
    resp := new(pb.HelloResponse)
    resp.Message = fmt.Sprintf("Hello %s.", in.Name)

    return resp, nil
}

func main() {
    listen, err := net.Listen("tcp", Address)
    if err != nil {
        grpclog.Fatalf("failed to listen: %v", err)
    }

    // 实例化grpc Server
    s := grpc.NewServer()

    // 注册HelloService
    pb.RegisterHelloServer(s, HelloService)

    // 开启trace
    go startTrace()

    grpclog.Println("Listen on " + Address)
    s.Serve(listen)
}

func startTrace() {
    trace.AuthRequest = func(req *http.Request) (any, sensitive bool) {
        return true, true
    }
    go http.ListenAndServe(":50051", nil)
    grpclog.Println("Trace listen on 50051")
}
```

这里我们开启一个http服务监听50051端口，用来查看grpc请求的trace信息

运行：

```
$ go run main.go

Listen on 127.0.0.1:50052
Trace listen on 50051

# 进入client目录执行一次客户端请求
```

服务端事件查看

访问：localhost:50051/debug/events，结果如图：

可以看到服务端注册的服务和服务正常启动的事件信息。

请求日志信息查看

访问：localhost:50051/debug/requests，结果如图：

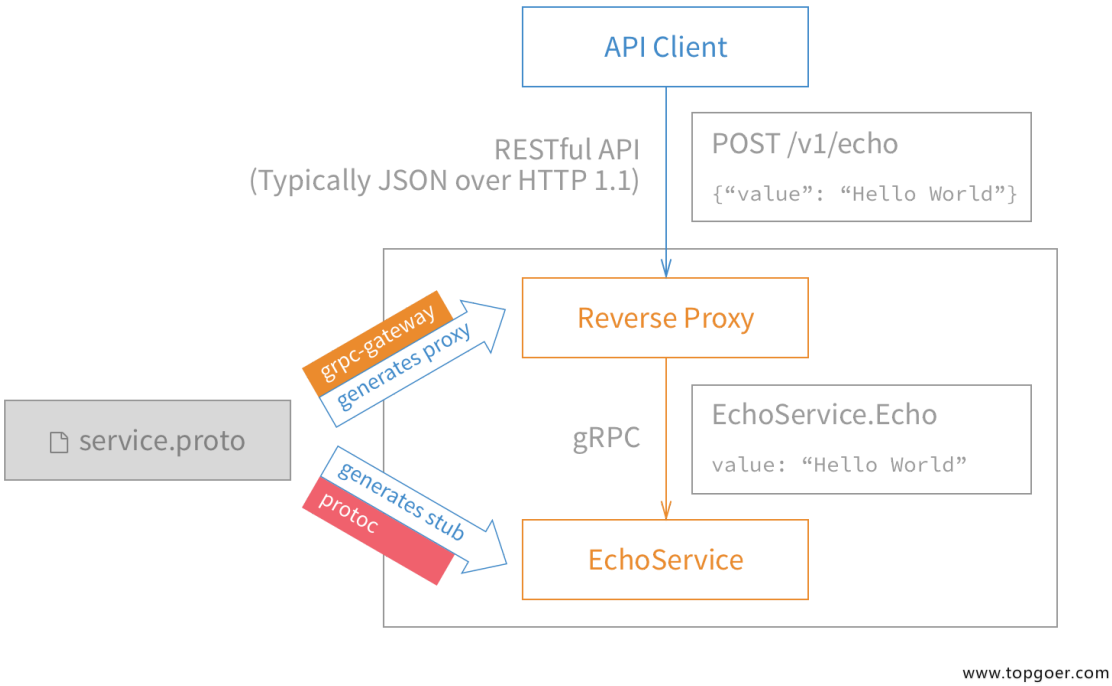
这里可以显示最近的请求状态，包括请求的服务、参数、耗时、响应，对于简单的状态查看还是很方便的，默认值显示最近10条记录。

HTTP网关

源自coreos的一篇博客 Take a REST with HTTP/2, Protobufs, and Swagger。

etcd3 API全面升级为gRPC后，同时要提供REST API服务，维护两个版本的服务显然不太合理，所以grpc-gateway诞生了。通过protobuf的自定义option实现了一个网关，服务端同时开启gRPC和HTTP服务，HTTP服务接收客户端请求后转换为grpc请求数据，获取响应后转为json数据返回给客户端。

结构如图：



安装grpc-gateway

```
$ go get -u github.com/grpc-ecosystem/grpc-gateway/protoc-gen-grpc-gateway
```

目录结构

```
|—— hello_http/
    |—— client/
        |—— main.go    // 客户端
    |—— server/
        |—— main.go    // GRPC服务端
```

```
    |—— server_http/
        |—— main.go    // HTTP服务端
|—— proto/
    |—— google        // googleApi http-proto定义
        |—— api
            |—— annotations.proto
            |—— annotations.pb.go
            |—— http.proto
            |—— http.pb.go
    |—— hello_http/
        |—— hello_http.proto    // proto描述文件
        |—— hello_http.pb.go    // proto编译后文件
        |—— hello_http_pb.gw.go // gateway编译后文件
```

这里用到了google官方Api中的两个proto描述文件，直接拷贝不要做修改，里面定义了protocol buffer扩展的HTTP option，为grpc的http转换提供支持。

示例代码

Step 1. 编写proto描述文件：proto/hello_http.proto

```
syntax = "proto3";

package hello_http;
option go_package = "hello_http";

import "google/api/annotations.proto";

// 定义Hello服务
service HelloHTTP {
    // 定义SayHello方法
    rpc SayHello(HelloHTTPRequest) returns (HelloHTTPResponse) {
        // http option
        option (google.api.http) = {
            post: "/example/echo"
            body: "*"
        };
    }
}

// HelloRequest 请求结构
message HelloHTTPRequest {
    string name = 1;
}
```

```
// HelloResponse 响应结构
message HelloHTTPResponse {
    string message = 1;
}
```

这里在原来的 `SayHello` 方法定义中增加了http option, POST方式，路由为"/example/echo"。

Step 2. 编译proto

```
$ cd proto

# 编译google.api
$ protoc -I . --go_out=plugins=grpc,Mgoogle/protobuf/descriptor.proto=github.com/golang/protobuf/protoc-gen-go/descriptor:. google/api/*.proto

# 编译hello_http.proto
$ protoc -I . --go_out=plugins=grpc,Mgoogle/api/annotations.proto=github.com/jergoo/go-grpc-example/proto/google/api:. hello_http/*.proto

# 编译hello_http.proto gateway
$ protoc --grpc-gateway_out=logtostderr=true:. hello_http/hello_http.proto
```

注意这里需要编译google/api中的两个proto文件，同时在编译hello_http.proto时使用 `M` 参数指定引入包名，最后使用grpc-gateway编译生成 `hello_http_pb.gw.go` 文件，这个文件就是用来做协议转换的，查看文件可以看到里面生成的http handler，处理proto文件中定义的路由"example/echo"接收POST参数，调用HelloHTTP服务的客户端请求grpc服务并响应结果。

Step 3: 实现服务端和客户端

server/main.go和client/main.go的实现与hello项目一致，这里不再说明。

> server_http/main.go

```
package main

import (
    "net/http"

    "github.com/grpc-ecosystem/grpc-gateway/runtime"
    "golang.org/x/net/context"
    "google.golang.org/grpc"
    "google.golang.org/grpc/grpclog"
```

```
    gw "github.com/jergoo/go-grpc-example/proto/hello_http"
)

func main() {
    ctx := context.Background()
    ctx, cancel := context.WithCancel(ctx)
    defer cancel()

    // grpc服务地址
    endpoint := "127.0.0.1:50052"
    mux := runtime.NewServeMux()
    opts := []grpc.DialOption{grpc.WithInsecure()}

    // HTTP转grpc
    err := gw.RegisterHelloHTTPHandlerFromEndpoint(ctx, mux, endpoint, opts)
    if err != nil {
        grpclog.Fatalf("Register handler err:%v\n", err)
    }

    grpclog.Println("HTTP Listen on 8080")
    http.ListenAndServe(":8080", mux)
}
```

就是这么简单。开启了一个http server，收到请求后根据路由转发请求到对应的RPC接口获得结果。grpc-gateway做的事情就是帮我们自动生成了转换过程的实现。

运行结果

依次开启gRPC服务和HTTP服务端：

```
$ cd hello_http/server && go run main.go
Listen on 127.0.0.1:50052
```

```
$ cd hello_http/server_http && go run main.go
HTTP Listen on 8080
```

调用grpc客户端：

```
$ cd hello_http/client && go run main.go
Hello gRPC.

# HTTP 请求
```

```
$ curl -X POST -k http://localhost:8080/example/echo -d '{"name": "gRPC-HTTP is working!"}'
{"message":"Hello gRPC-HTTP is working!."}
```

升级版服务端

上面的使用方式已经实现了我们最初的需求，grpc-gateway项目中提供的示例也是这种使用方式，这样后台需要开启两个服务两个端口。其实我们也可以只开启一个服务，同时提供http和gRPC调用方式。

新建一个项目 `hello_http_2` ，基于 `hello_tls` 项目改造。客户端只要修改调用的proto包地址就可以了，这里我们看服务端的实现：

> hello_http_2/server/main.go

```go
package main

import (
    "crypto/tls"
    "io/ioutil"
    "net"
    "net/http"
    "strings"

    "github.com/grpc-ecosystem/grpc-gateway/runtime"
    pb "github.com/jergoo/go-grpc-example/proto/hello_http"
    "golang.org/x/net/context"
    "golang.org/x/net/http2"
    "google.golang.org/grpc"
    "google.golang.org/grpc/credentials"
    "google.golang.org/grpc/grpclog"
)

// 定义helloHTTPService并实现约定的接口
type helloHTTPService struct{}

// HelloHTTPService Hello HTTP服务
var HelloHTTPService = helloHTTPService{}

// SayHello 实现Hello服务接口
func (h helloHTTPService) SayHello(ctx context.Context, in *pb.HelloHTTPRequest)
(*pb.HelloHTTPResponse, error) {
    resp := new(pb.HelloHTTPResponse)
    resp.Message = "Hello " + in.Name + "."
```

```
    return resp, nil
}

func main() {
    endpoint := "127.0.0.1:50052"
    conn, err := net.Listen("tcp", endpoint)
    if err != nil {
        grpclog.Fatalf("TCP Listen err:%v\n", err)
    }

    // grpc tls server
    creds, err := credentials.NewServerTLSFromFile("../../keys/server.pem",
"../../keys/server.key")
    if err != nil {
        grpclog.Fatalf("Failed to create server TLS credentials %v", err)
    }
    grpcServer := grpc.NewServer(grpc.Creds(creds))
    pb.RegisterHelloHTTPServer(grpcServer, HelloHTTPService)

    // gw server
    ctx := context.Background()
    dcreds, err := credentials.NewClientTLSFromFile("../../keys/server.pem", "se
rver name")
    if err != nil {
        grpclog.Fatalf("Failed to create client TLS credentials %v", err)
    }
    dopts := []grpc.DialOption{grpc.WithTransportCredentials(dcreds)}
    gwmux := runtime.NewServeMux()
    if err = pb.RegisterHelloHTTPHandlerFromEndpoint(ctx, gwmux, endpoint, dopt
s); err != nil {
        grpclog.Fatalf("Failed to register gw server: %v\n", err)
    }

    // http服务
    mux := http.NewServeMux()
    mux.Handle("/", gwmux)

    srv := &http.Server{
        Addr:      endpoint,
        Handler:   grpcHandlerFunc(grpcServer, mux),
        TLSConfig: getTLSConfig(),
    }

    grpclog.Infof("gRPC and https listen on: %s\n", endpoint)
```

```
    if err = srv.Serve(tls.NewListener(conn, srv.TLSConfig)); err != nil {
        grpclog.Fatal("ListenAndServe: ", err)
    }

    return
}

func getTLSConfig() *tls.Config {
    cert, _ := ioutil.ReadFile("../../keys/server.pem")
    key, _ := ioutil.ReadFile("../../keys/server.key")
    var demoKeyPair *tls.Certificate
    pair, err := tls.X509KeyPair(cert, key)
    if err != nil {
        grpclog.Fatalf("TLS KeyPair err: %v\n", err)
    }
    demoKeyPair = &pair
    return &tls.Config{
        Certificates: []tls.Certificate{*demoKeyPair},
        NextProtos:   []string{http2.NextProtoTLS}, // HTTP2 TLS支持
    }
}

// grpcHandlerFunc returns an http.Handler that delegates to grpcServer on incom
ing gRPC
// connections or otherHandler otherwise. Copied from cockroachdb.
func grpcHandlerFunc(grpcServer *grpc.Server, otherHandler http.Handler) http.Ha
ndler {
    if otherHandler == nil {
        return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
            grpcServer.ServeHTTP(w, r)
        })
    }
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        if r.ProtoMajor == 2 && strings.Contains(r.Header.Get("Content-Type"),
"application/grpc") {
            grpcServer.ServeHTTP(w, r)
        } else {
            otherHandler.ServeHTTP(w, r)
        }
    })
}
```

gRPC服务端接口的实现没有区别，重点在于HTTP服务的实现。gRPC是基于http2实现的， `net/http` 包也实现了http2，所以我们可以开启一个HTTP服务同时服务两个版本的协议，在注册http handler的时候，在方法 `grpcHandlerFunc` 中检测请求头信息，决定是

直接调用gRPC服务，还是使用gateway的HTTP服务。 `net/http` 中对http2的支持要求开启https，所以这里要求使用https服务。

步骤

- 注册开启TLS的grpc服务
- 注册开启TLS的gateway服务，地址指向grpc服务
- 开启HTTP server

运行结果

```
$ cd hello_http_2/server && go run main.go
gRPC and https listen on: 127.0.0.1:50052
```

```
$ cd hello_http_2/client && go run main.go
Hello gRPC.

# HTTP 请求
$ curl -X POST -k https://localhost:50052/example/echo -d '{"name": "gRPC-HTTP is working!"}'
{"message":"Hello gRPC-HTTP is working!."}
```