

微服务框架开发二

联系QQ: 2816010068, 加入会员群

目录

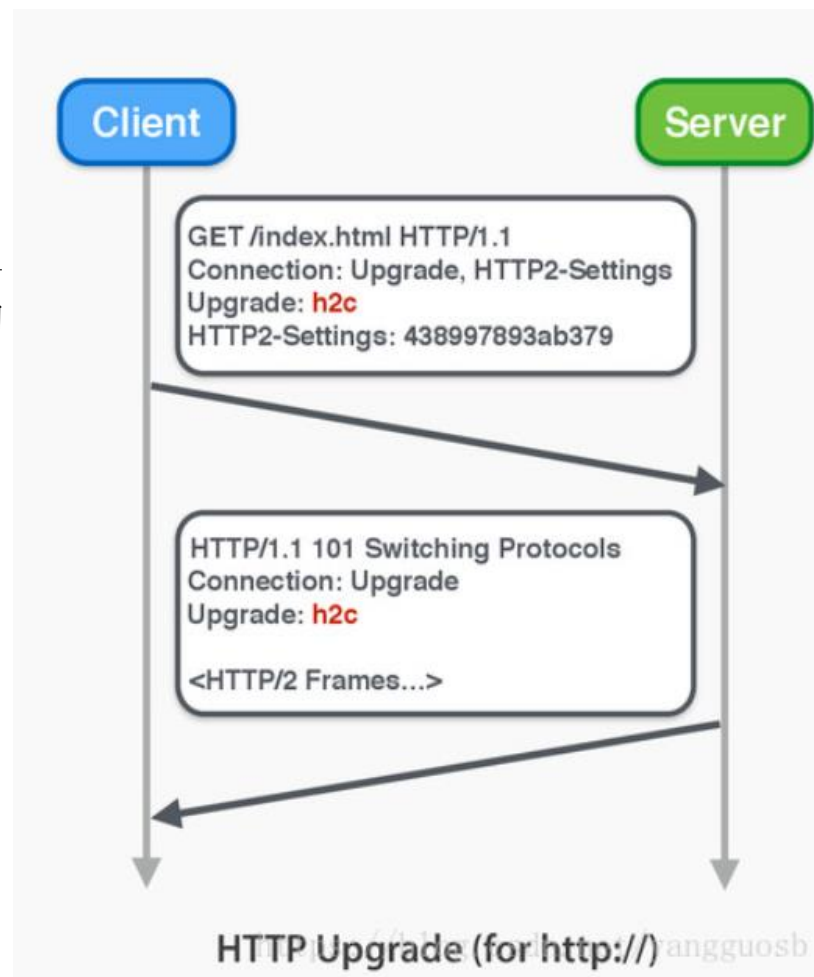
- Http2 协议协商
- https 密码交换协商
- grpc使用示例

http2使用示例

- Golang默认支持http2
 - 问题?
 - 如何兼容老的浏览器或客户端
 - 通过协议协商进行解决
 - 协议协商
 - Upgrade机制
 - ALPN机制, Application Layer Protocol Negotiation

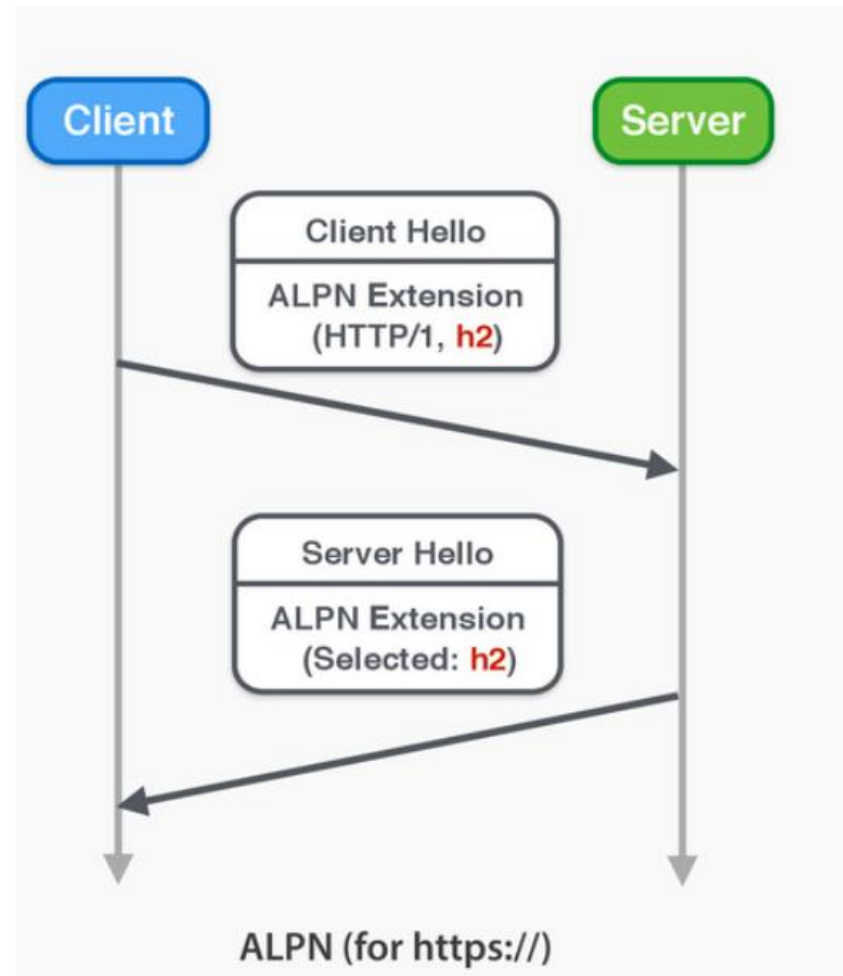
http2协议协商机制

- Upgrade机制
 - 客户端主动发起
 - 如果服务端支持http2，返回101switching protocol
 - 如果服务端不支持http2，直接忽略，返回http1.1的内容
 - 版本标识符
 - 字符串 “h2c” 标示运行在明文 TCP 之上的 HTTP/2 协议（http模式）；
 - 字符串 “h2” 标示使用了 TLS的 HTTP/2 协议（https模式）
 - Chrome和firefox，只支持加密（tls）的http2



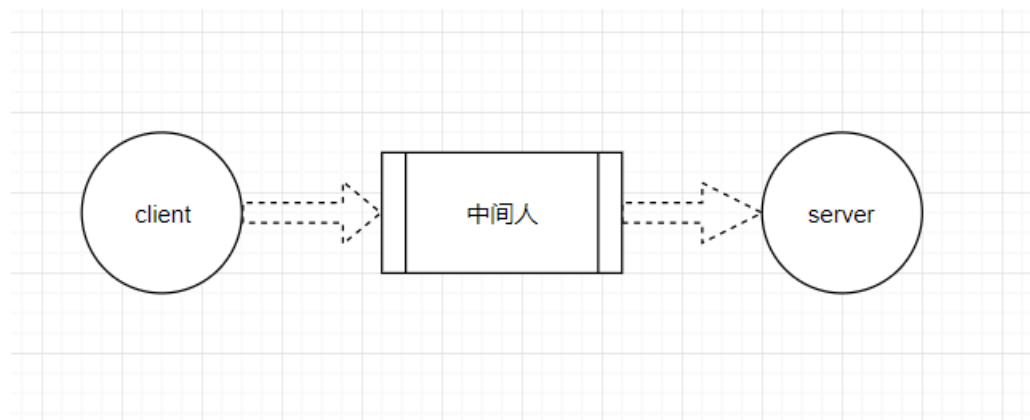
http2协议协商机制

- ALPN机制, (Application Layer Protocol Negotiation)
 - 客户端主动发起
 - 如果服务端支持http2, 则返回selected h2
 - 如果服务端不支持http2, 则返回selected http1.1
 - 协商的时机
 - 在https交换加密密钥的过程中进行协商
 - ALPN只应用于https的http2.0



http协议

- 明文传输
 - 信息不安全
 - 容易被截取
 - 容易被篡改

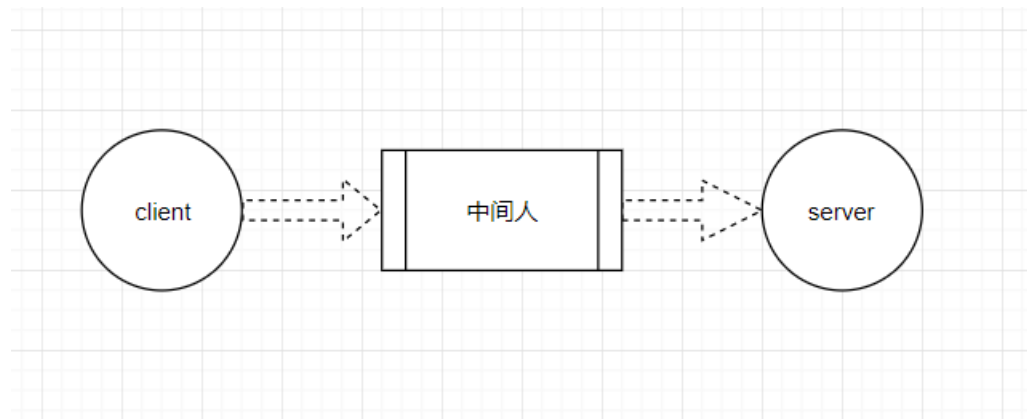


https密钥交换过程

- 加密方式
 - 对称加密
 - 用来加密和解密的密钥相同
 - 优点：加密性能好
 - 缺点：密钥管理问题
 - 非对称加密
 - 用来加解密的密钥不一样
 - 优点：安全性非常高
 - 缺点：加密性能比较差

http协议-对称加密

- 对称加密是否解决问题
 - 交换密钥
 - 发送方，使用密钥对内容进行加密
 - 接收方，使用密钥对内容估进行解密



https密钥交换过程

- 非对称加密算法
 - 私钥
 - 需要进行保密，不能公开
 - 私钥加密的内容，只能通过公钥进行解密。通过私钥和加密算法，能够推导出公钥
 - 公钥
 - 公钥是公开的，通过公钥和加密算法，不能够推导出私钥
 - 公钥加密的内容，只能通过私钥才能解密。

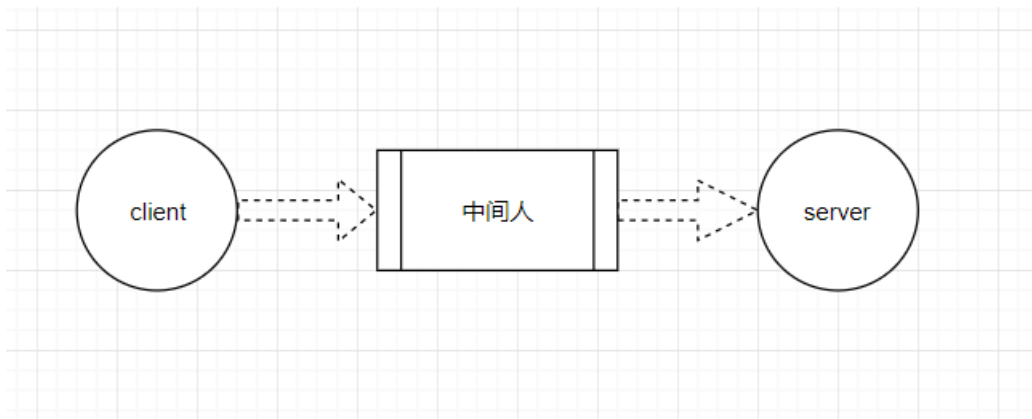
https密钥交换过程

- 密钥协商

- 浏览器使用Https的URL访问服务器，建立SSL链接。
- 服务器接收到SSL链接后，发送非对称加密的公钥A给浏览器。
- 浏览器生成随机数，作为对称加密的密钥B。
- 浏览器使用服务器返回的公钥A，对自己生成的对称加密密钥B进行加密，得到密钥C。
- 浏览器将密钥C发送给服务器
- 服务器使用自己的私钥D对接受的密钥C进行解密，得到对称加密密钥B。
- 浏览器和服务器之间使用密钥B作为对称加密密钥进行通信
- 优点： 非对称加密只使用了一次，后续所有的通信消息都是用对称加密，效率比非对称加密高。

https密钥交换

- 中间人攻击
 - 当服务器发送公钥给客户端， 中间人截获公钥 ；
 - 将 中间人自己的公钥 冒充服务器的公钥发送给客户端；
 - 之后客户端会用 中间人的公钥 来加密自己生成的 对称密钥 。
 - 然后把加密的密钥发送给服务器，这时中间人又把密钥截取；
 - 中间人用自己的私钥把加密的密钥进行解密，解密后中间人就能获取 对称加密的密钥 。
 - 注意： 非对称加密之所以不安全，因为客户端不知道这把公钥是不是属于服务器的。



https 密钥交换

- 数字证书
 - 认证中心(CA):
 - 一个拥有公信力、大家都认可的认证中心, 数字证书认证机构。
 - 证书内容:
 - 签发者
 - 证书用途
 - 公钥
 - 加密算法
 - Hash算法
 - 证书到期时间

https 密钥交换

- 基于数字证书的安全体系
 - 服务器在给客户端传输公钥的过程中，会把公钥和服务器的个人信息通过 hash算法生成信息摘要 。



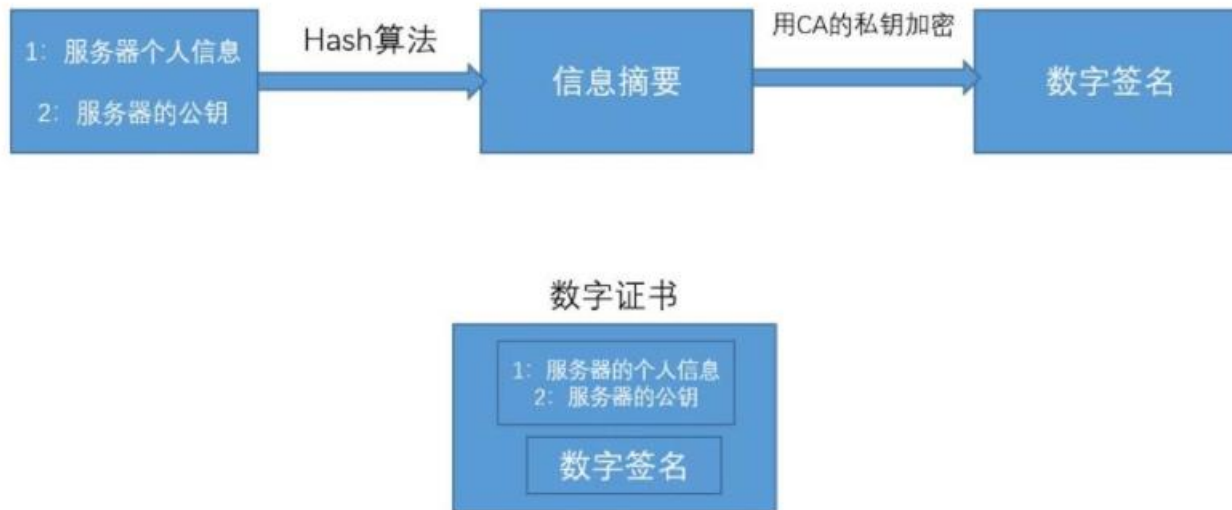
https 密钥交换

- 基于数字证书的安全体系
 - 为了防止信息摘要被调换，服务器会采用 CA提供的私钥信息摘要进行加密来形成数字签名。



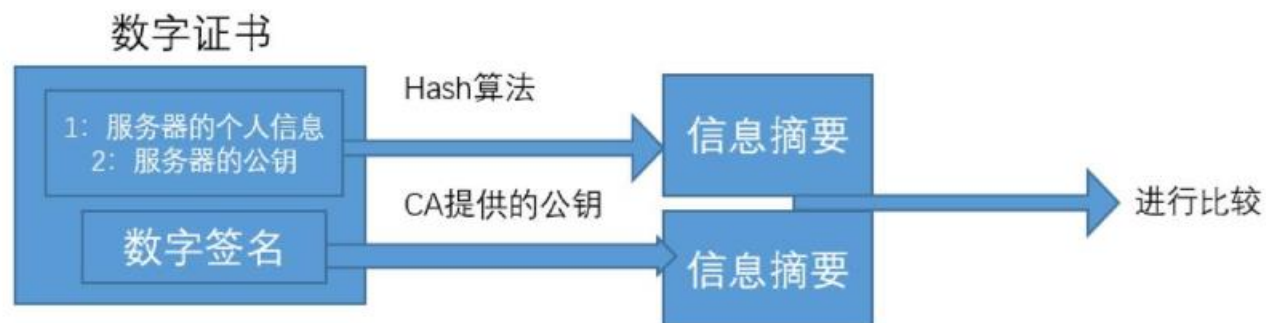
https 密钥交换

- 基于数字证书的安全体系
 - 最后会把原来没 Hash算法 之前的 个人信息、公钥及、数字签名 合并在一起，形成 数字证书。



https 密钥交换

- 基于数字证书的安全体系
 - 客户端拿到 数字证书 之后，使用 CA提供的公钥 对数字证书里的数字签名进行解密来得到信息摘要，然后对数字证书里服务器的公钥及个人信息进行Hash得到另一份信息摘要。



Reference: 3rd Edition, Network Security, 15-221/2452/PS1

https 密钥交换

- 常见问题

常见：证书中包含了host信息，假如客户端正在访问的host和证书中的host不一样，浏览器会发出警告。



https 密钥交换

- 证书颁发

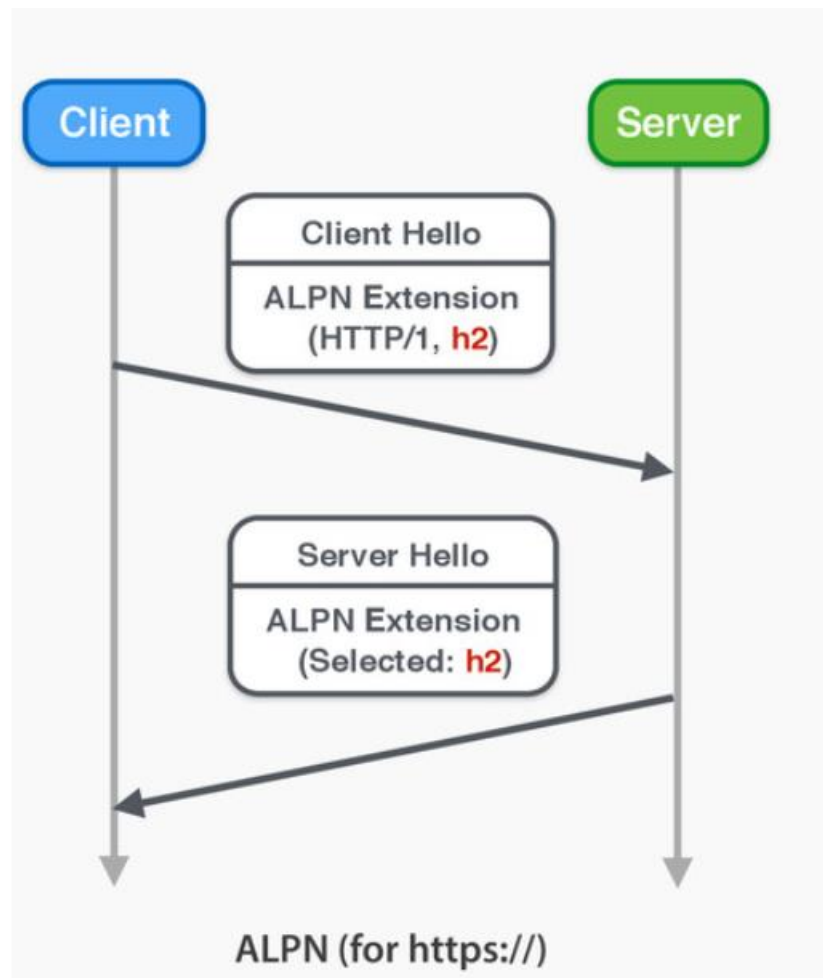
CA证书的颁发

服务器一开始向认证中心申请证书，客户端也内置这些证书。

当客户端收到服务器传输过来的数据数字证书时，就会在内置的证书列表里，查看是否有解开该数字证书的公钥，如果有则认证，如果没有则不认证。

https 密钥交换

- ALPN协商，探测服务端是否支持http2



Grpc使用示例

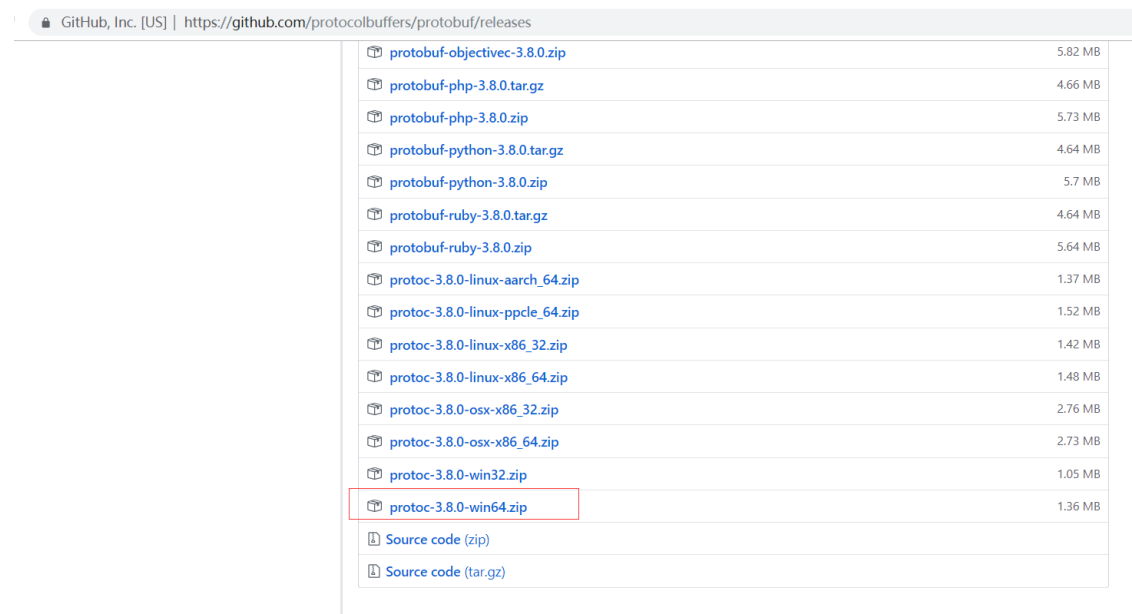
- 编写IDL，通过IDL来定义我们的服务，保存为hello.proto


















```
syntax = "proto3";  
package hello;  
  
message HelloRequest {  
    string name = 1;  
}  
  
message HelloResponse {  
    string reply = 1;  
}  
  
service HelloService {  
    rpc SayHello(HelloRequest) returns (HelloResponse){}  
}
```

Grpc使用示例

- 代码生成工具安装

- 安装protoc工具，下载地址：<https://github.com/protocolbuffers/protobuf/releases>
- 安装golang扩展，`go get -u github.com/golang/protobuf/protoc-gen-go`
- 安装grpc运行库，`go get google.golang.org/grpc`



GitHub, Inc. [US] https://github.com/protocolbuffers/protobuf/releases	
 protobuf-objectivec-3.8.0.zip	5.82 MB
 protobuf-php-3.8.0.tar.gz	4.66 MB
 protobuf-php-3.8.0.zip	5.73 MB
 protobuf-python-3.8.0.tar.gz	4.64 MB
 protobuf-python-3.8.0.zip	5.7 MB
 protobuf-ruby-3.8.0.tar.gz	4.64 MB
 protobuf-ruby-3.8.0.zip	5.64 MB
 protoc-3.8.0-linux-aarch_64.zip	1.37 MB
 protoc-3.8.0-linux-ppc64_64.zip	1.52 MB
 protoc-3.8.0-linux-x86_32.zip	1.42 MB
 protoc-3.8.0-linux-x86_64.zip	1.48 MB
 protoc-3.8.0-osx-x86_32.zip	2.76 MB
 protoc-3.8.0-osx-x86_64.zip	2.73 MB
 protoc-3.8.0-win32.zip	1.05 MB
 protoc-3.8.0-win64.zip	1.36 MB
 Source code (zip)	
 Source code (tar.gz)	

Grpc使用示例

- 生成代码
 - `protoc --go_out=plugins=grpc:. hello.proto`

Grpc使用示例

- 实现业务逻辑

```
const (  
    port = ":50051"  
)  
  
type server struct{  
  
func (s *server) SayHello(ctx context.Context, in *pb.HelloRequest) (*pb.HelloResponse, error) {  
    return &pb.HelloResponse{Reply: "你好 |" + in.Name}, nil  
}  
  
func main() {  
    lis, err := net.Listen("tcp", port)  
    if err != nil {  
        log.Fatal("failed to listen: %v", err)  
    }  
    s := grpc.NewServer()  
    pb.RegisterHelloServiceServer(s, &server{})  
    s.Serve(lis)  
}
```

Grpc使用示例

- 实现客户端代码

```
func main() {
    conn, err := grpc.Dial(address, grpc.WithInsecure())
    if err != nil {
        log.Fatal("did not connect: %v", err)
    }
    defer conn.Close()
    c := pb.NewHelloServiceClient(conn)

    name := defaultName
    if len(os.Args) > 1 {
        name = os.Args[1]
    }
    r, err := c.SayHello(context.Background(), &pb.HelloRequest{Name: name})
    if err != nil {
        log.Fatal("could not greet: %v", err)
    }
    log.Printf("Greeting: %s", r.Reply)
}
```