Go RPC & TLS 鉴权简明教程

Go 简明教程系列文章链接：

- Go 语言简明教程 (Aug 6, 2019)
- Go Gin 简明教程 (Aug 7, 2019)
- Go2 新特性简明教程 (Aug 15, 2019)
- Go Protobuf 简明教程 (Jan 11, 2020)
- Go RPC & TLS 鉴权简明教程 (Jan 13, 2020)
- Go WebAssembly (Wasm) 简明教程 (Jan 23, 2020)
- Go Test 单元测试简明教程 (Feb 10, 2020)
- Go Mock (gomock)简明教程 (Feb 14, 2020)
- Go Mmap 文件内存映射简明教程 (Apr 20, 2020)
- Go Context 并发编程简明教程 (Apr 20, 2020)

本文介绍了 Go 语言远程过程调用(Remote Procedure Call, RPC)的使用方式，示例基于 Golang 标准库 net/rpc，同时介绍了如何基于 TLS/SSL 实现服务器端和客户端的单向鉴权、双向鉴权。

1 RPC 简介

> 远程过程调用（英语：Remote Procedure Call，缩写为 RPC）是一个计算机通信协议。该协议允许运行于一台计算机的程序调用另一个地址空间（通常为一个开放网络的一台计算机）的子程序，而程序员就像调用本地程序一样，无需额外地为这个交互作用编程（无需关注细节）。RPC是一种服务器-客户端（Client/Server）模式，经典实现是一个通过发送请求-接受回应进行信息交互的系统。
> – 远程过程调用 - Wikipedia.org

划重点：**程序员就像调用本地程序一样，无需关注细节**

RPC 协议假定某种传输协议(TCP, UDP)存在，为通信程序之间携带信息数据。使用 RPC 协议，无需关注底层网络技术协议，调用远程方法就像在调用本地方法一样。

RPC 流程：

RPC 模型是一个典型的客户端-服务器模型(Client-Server, CS)，相比于调用本地的接口，RPC 还需要知道的是服务器端的地址信息。本地调用，好比两个人面对面说话，而 RPC 好比打电话，需要知道对方的电话号码，但是并不需要关心语音是怎么编码，如何传输，又如何解码的。

接下来我们将展示如何将一个简单的本地调用的程序一步步地改造一个 RPC 服务。

2 一个简单的计算二次方的程序

不考虑 RPC 调用，仅考虑本地调用的场景，程序实现如下：

```go
// main.go
package main

import "log"

type Result struct {
    Num, Ans int
}

type Cal int

func (cal *Cal) Square(num int) *Result {
    return &Result{
        Num: num,
        Ans: num * num,
    }
}

func main() {
    cal := new(Cal)
    result := cal.Square(12)
    log.Printf("%d^2 = %d", result.Num, result.Ans)
}
```

在这个20行的程序中，我们做了以下几件事：

- `Cal` 结构体，提供了 Square 方法，用于计算传入参数 num 的 二次方。
- `Result` 结构体，包含 Num 和 Ans 两个字段，Ans 是计算后的值，Num 是待计算的值。
- `main` 函数，测试我们实现的 Square 方法。

运行 main.go，将会输出

```
$ go run main.go
2020/01/13 20:27:08 12^2 = 144
```

3 RPC 需要满足什么条件

net/rpc，方法需要长这个样子：

```
func (t *T) MethodName(argType T1, replyType *T2) error
```

即需要满足以下 5 个条件：

- 1. 方法类型（T）是导出的（首字母大写）
- 2. 方法名（MethodName）是导出的
- 3. 方法有2个参数(argType T1, replyType *T2)，均为导出/内置类型
- 4. 方法的第2个参数一个指针(replyType *T2)
- 5. 方法的返回值类型是 error

net/rpc 对参数个数的限制比较严格，仅能有2个，第一个参数是调用者提供的请求参数，第二个参数是返回给调用者的响应参数，也就是说，服务端需要将计算结果写在第二个参数中。如果调用过程中发生错误，会返回 error 给调用者。

接下来，我们改造下 Square 函数，以满足上述 5 个条件。

```
func (cal *Cal) Square(num int, result *Result) error {
    result.Num = num
    result.Ans = num * num
    return nil
}

func main() {
    cal := new(Cal)
    var result Result
    cal.Square(11, &result)
    log.Printf("%d^2 = %d", result.Num, result.Ans)
}
```

- Cal 和 Square 均为导出类型，满足条件 1) 和 2)
- 2 个参数， num int 为内置类型， result *Result 为导出类型，满足条件 3)
- 第2个参数 result *Result 是一个指针，满足条件 4)
- 返回值类型是 error，满足条件 5)

至此，方法 Cal.Square 满足了 RPC 调用的5个条件。

4 RPC 服务与调用

4.1 基于HTTP，启动 RPC 服务

待 HTTP 请求。

接下来我们新建一个文件夹 server，将 Cal.Square 方法移动到 server/main.go 中，并在 main 函数中启动 RPC 服务。

```go
// server/main.go
package main

import (
    "log"
    "net"
    "net/http"
    "net/rpc"
)

type Result struct {
    Num, Ans int
}

type Cal int

func (cal *Cal) Square(num int, result *Result) error {
    result.Num = num
    result.Ans = num * num
    return nil
}

func main() {
    rpc.Register(new(Cal))
    rpc.HandleHTTP()

    log.Printf("Serving RPC server on port %d", 1234)
    if err := http.ListenAndServe(":1234", nil); err != nil {
        log.Fatal("Error serving: ", err)
    }
}
```

- 使用 `rpc.Register`，发布 Cal 中满足 RPC 注册条件的方法（Cal.Square）
- 使用 `rpc.HandleHTTP` 注册用于处理 RPC 消息的 HTTP Handler
- 使用 `http.ListenAndServe` 监听 1234 端口，等待 RPC 请求。

我们在 server 目录下，执行

此时，RPC 服务已经启动，等待客户端的调用。

4.2 实现客户端

我们在 client 目录中新建文件 client/main.go，创建 HTTP 客户端，调用 Cal.Square 方法。

```
1  // client/main.go
2  package main
3
4  import (
5      "log"
6      "net/rpc"
7  )
8
9  type Result struct {
10     Num, Ans int
11 }
12
13 func main() {
14     client, _ := rpc.DialHTTP("tcp", "localhost:1234")
15     var result Result
16     if err := client.Call("Cal.Square", 12, &result); err != nil {
17         log.Fatal("Failed to call Cal.Square. ", err)
18     }
19     log.Printf("%d^2 = %d", result.Num, result.Ans)
20 }
```

在客户端的实现中，因为要用到 Result 类型，简单起见，我们拷贝了 `Result` 的定义。

- 使用 `rpc.DialHTTP` 创建了 HTTP 客户端 client，并且创建了与 localhost:1234 的链接，1234 恰好是 RPC 服务监听的端口。
- 使用 `rpc.Call` 调用远程方法，第1个参数是方法名 Cal.Square，后两个参数与 Cal.Square 的定义的参数相对应。

我们在 client 目录下，执行

```
1  2020/01/13 21:17:45 12^2 = 144
```

如果能够返回计算的结果，说明调用成功。

4.3 异步调用

```
func main() {
    client, _ := rpc.DialHTTP("tcp", "localhost:1234")
    var result Result
    asyncCall := client.Go("Cal.Square", 12, &result, nil)
    log.Printf("%d^2 = %d", result.Num, result.Ans)

    <-asyncCall.Done
    log.Printf("%d^2 = %d", result.Num, result.Ans)

}
```

执行结果如下：

```
2020/01/13 21:34:26 0^2 = 0
2020/01/13 21:34:26 12^2 = 144
```

因为 `client.Go` 是异步调用，因此第一次打印 result，result 没有被赋值。而通过调用 `<-asyncCall.Done`，阻塞当前程序直到 RPC 调用结束，因此第二次打印 result 时，能够看到正确的赋值。

5 证书鉴权(TLS/SSL)

5.1 客户端对服务器端鉴权

HTTP 协议默认是不加密的，我们可以使用证书来保证通信过程的安全。

生成私钥和自签名的证书，并将 server.key 权限设置为只读，保证私钥的安全。

```
# 生成私钥
openssl genrsa -out server.key 2048
# 生成证书
openssl req -new -x509 -key server.key -out server.crt -days 3650
# 只读权限
chmod 400 server.key
```

执行完，当前文件夹下多出了 server.crt 和 server.key 2 个文件。

服务器端可以使用生成的 server.crt 和 server.key 文件启动 TLS 的端口监听。

```
// server/main.go
import (
    "crypto/tls"
```

```
)

func main() {
    rpc.Register(new(Cal))
    cert, _ := tls.LoadX509KeyPair("server.crt", "server.key")
    config := &tls.Config{
        Certificates: []tls.Certificate{cert},
    }
    listener, _ := tls.Listen("tcp", ":1234", config)
    log.Printf("Serving RPC server on port %d", 1234)

    for {
        conn, _ := listener.Accept()
        defer conn.Close()
        go rpc.ServeConn(conn)
    }
}
```

客户端也需要做相应的修改，使用 `tls.Dial` 代替 `rpc.DialHTTP` 连接服务端，如果客户端不需要对服务端鉴权，那么可以设置 `InsecureSkipVerify:true`，即可跳过对服务端的鉴权，例如：

```
// client/main.go
import (
    "crypto/tls"
    "log"
    "net/rpc"
)

func main() {
    config := &tls.Config{
        InsecureSkipVerify: true,
    }
    conn, _ := tls.Dial("tcp", "localhost:1234", config)
    defer conn.Close()
    client := rpc.NewClient(conn)

    var result Result
    if err := client.Call("Cal.Square", 12, &result); err != nil {
        log.Fatal("Failed to call Cal.Square. ", err)
    }

```

如果需要对服务器端鉴权，那么需要将服务端的证书添加到信任证书池中，如下：

```
// client/main.go

func main() {
    certPool := x509.NewCertPool()
    certBytes, err := ioutil.ReadFile("../server/server.crt")
    if err != nil {
        log.Fatal("Failed to read server.crt")
    }
    certPool.AppendCertsFromPEM(certBytes)

    config := &tls.Config{
        RootCAs: certPool,
    }

    conn, _ := tls.Dial("tcp", "localhost:1234", config)
    defer conn.Close()
    client := rpc.NewClient(conn)

    var result Result
    if err := client.Call("Cal.Square", 12, &result); err != nil {
        log.Fatal("Failed to call Cal.Square. ", err)
    }

    log.Printf("%d^2 = %d", result.Num, result.Ans)
}
```

5.2 服务器端对客户端的鉴权

服务器端对客户端的鉴权是类似的，核心在于 `tls.Config` 的配置：

- 把对方的证书添加到自己的信任证书池 `RootCAs` (客户端配置)， `ClientCAs` (服务器端配置) 中。
- 创建链接时，配置自己的证书 `Certificates` 。

客户端的 config 作如下修改：

```
// client/main.go

cert, _ := tls.LoadX509KeyPair("client.crt", "client.key")
certPool := x509.NewCertPool()
```

```
config := &tls.Config{
    Certificates: []tls.Certificate{cert},
    RootCAs: certPool,
}
```

服务器端的 config 作如下修改：

```
// server/main.go

cert, _ := tls.LoadX509KeyPair("server.crt", "server.key")
certPool := x509.NewCertPool()
certBytes, _ := ioutil.ReadFile("../client/client.crt")
certPool.AppendCertsFromPEM(certBytes)
config := &tls.Config{
    Certificates: []tls.Certificate{cert},
    ClientAuth:   tls.RequireAndVerifyClientCert,
    ClientCAs:    certPool,
}
```

附：参考

1. Golang net/rpc 官方文档 - golang.org
2. Golang TLS 配置 - github.com

专题: Go 简明教程

本文发表于 2020-01-13，最后修改于 2021-02-06。

本站永久域名「 geektutu.com 」，也可搜索「 极客兔兔 」找到我。

上一篇 « Go Protobuf 简明教程 下一篇 » Go WebAssembly (Wasm) 简明教程

推荐阅读

动手写ORM框架 - GeeORM第六天 支持事务(Transaction)

博客折腾记(五) - 友链这件事，没那么简单

发表于2019-07-03， 阅读约13分钟

博客折腾记(三) - 主题设计、彩蛋与阅读量翻倍

发表于2019-06-23， 阅读约14分钟

#关于我 (9) #Go (48) #百宝箱 (2) #Cheat Sheet (1) #Go语言高性能编程 (20) #友链 (1) #Pandas (3)

#机器学习 (9) #TensorFlow (9) #mnist (5) #Python (10) #强化学习 (3) #OpenAI gym (4) #DQN (1)

#Q-Learning (1) #CNN (1) #TensorFlow 2 (10) #官方文档 (10) #Rust (1)

去 Github 评论