Go语言微服务架构实战：第十六节 gRPC框架--RPC简介及编程


Davie
Java开发 | Android开发 | Go语言编程资深爱好者
关注他

RPC入门（Remote Procedure Call Protocol）

RPC是远程过程调用的简称，是分布式系统中不同节点间流行的通信方式。在互联网时代，RPC已经和IPC一样成为一个不可或缺的基础构件。因此Go语言的标准库也提供了一个简单的RPC实现，我们将以此为入口学习RPC的各种用法。

Go语言的RPC包的路径为net/rpc，也就是放在了net包目录下面。因此可以猜测该RPC包是建立在net包基础之上的。下面尝试基于rpc实现一个Hello World的例子。

先构造一个HelloService类型，其中的Hello方法用于实现打印功能：

```
type HelloService struct {}

func (p *HelloService) Hello(request string, reply *string) error {
    *reply = "hello:" + request
    return nil
}
```

其中Hello方法必须满足Go语言的RPC规则：方法只能有两个可序列化的参数，其中第二个参数是指针类型，并且返回一个error类型，同时必须是公开的方法。

然后就可以将HelloService类型的对象注册为一个RPC服务：

```
func main() {
    rpc.RegisterName("HelloService", new(HelloService))

    listener, err := net.Listen("tcp", ":1234")
    if err != nil {
        log.Fatal("ListenTCP error:", err)
    }

    conn, err := listener.Accept()
    if err != nil {
        log.Fatal("Accept error:", err)
    }
    rpc.ServeConn(conn)
}
```

其中rpc.Register函数调用会将对象类型中所有满足RPC规则的对象方法注册为RPC函数，所有注册的方法会放在“HelloService”服务空间之下。然后我们建立一个唯一的TCP链接，并且通过rpc.ServeConn函数在该TCP链接上为对方提供RPC服务。

下面是客户端请求HelloService服务的代码：

客户端

```
func main() {
    client, err := rpc.Dial("tcp", "localhost:1234")
    if err != nil {
        log.Fatal
    }
```

```
    var reply string
    err = client.Call("HelloService.Hello", "hello", &reply)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println(reply)
}
```

首选是通过rpc.Dial拨号RPC服务，然后通过client.Call调用具体的RPC方法。在调用client.Call时，第一个参数是用点号链接的RPC服务名字和方法名字，第二和第三个参数分别我们定义RPC方法的两个参数。

由这个例子可以看出RPC的使用其实非常简单。

更安全的RPC接口

在涉及RPC的应用中，作为开发人员一般至少有三种角色：首选是服务端实现RPC方法的开发人员，其次是客户端调用RPC方法的人员，最后也是最重要的是制定服务端和客户端RPC接口规范的设计人员。在前面的例子中我们为了简化将以上几种角色的工作全部放到了一起，虽然看似实现简单，但是不利于后期的维护和工作的切割。

如果要重构HelloService服务，第一步需要明确服务的名字和接口：

```
const HelloServiceName = "path/to/pkg.HelloService"

type HelloServiceInterface = interface {
    Hello(request string, reply *string) error
}

func RegisterHelloService(svc HelloServiceInterface) error {
    return rpc.RegisterName(HelloServiceName, svc)
}
```

我们将RPC服务的接口规范分为三个部分：首先是服务的名字，然后是服务要实现的详细方法列表，最后是注册该类型服务的函数。为了避免名字冲突，我们在RPC服务的名字中增加了包路径前缀（这个是RPC服务抽象的包路径，并非完全等价Go语言的包路径）。RegisterHelloService注册服务时，编译器会要求传入的对象满足HelloServiceInterface接口。

在定义了RPC服务接口规范之后，客户端就可以根据规范编写RPC调用的代码了：

```
func main() {
    client, err := rpc.Dial("tcp", "localhost:1234")
    if err != nil {
        log.Fatal("dialing:", err)
    }

    var reply string
    err = client.Call(HelloServiceName+".Hello", "hello", &reply)
    if err != nil {
        log.Fatal(err)
    }
}
```

其中唯一的变化是client.Call的第一个参数用`HelloServiceName+".Hello"代替了"HelloService.Hello"。然而通过client.Call函数调用RPC方法依然比较繁琐，同时参数的类型依然无法得到编译器提供的安全保障。

为了简化客户端用户调用RPC函数，我们在可以在接口规范部分增加对客户端的简单包装：

```
type HelloServiceClient struct {
    *rpc.Client
}

var _ HelloServiceInterface = (*HelloServiceClient)(nil)

func DialHelloService(network, address string) (*HelloServiceClient, error) {
    c, err := rpc.Dial(network, address)
    if err != nil {
        return nil, err
    }
    return &HelloServiceClient{Client: c}, nil
}

func (p *HelloServiceClient) Hello(request string, reply *string) error {
    return p.Client.Call(HelloServiceName+".Hello", request, reply)
}
```

我们在接口规范中针对客户端新增加了HelloServiceClient类型，该类型也必须满足HelloServiceInterface接口，这样客户端用户就可以直接通过接口对应的方法调用RPC函数。同时提供了一个DialHelloService方法，直接拨号HelloService服务。

基于新的客户端接口，我们可以简化客户端用户的代码：

```
func main() {
    client, err := DialHelloService("tcp", "localhost:1234")
    if err != nil {
        log.Fatal("dialing:", err)
    }

    var reply string
    err = client.Hello("hello", &reply)
    if err != nil {
        log.Fatal(err)
    }
}
```

现在客户端用户不用再担心RPC方法名字或参数类型不匹配等低级错误的发生。

最后是基于RPC接口规范编写真实的服务端代码：

```
type HelloService struct {}

func (p *HelloService) Hello(request string, reply *string) error {
    *reply = "hello:" + request
    return nil
}

func main() {
    RegisterHelloService(new(HelloService))

    listener, err := net.Listen("tcp", ":1234")
    if err != nil {
        log.Fatal("ListenTCP error:", err)
    }

    for {
        conn, err := listener.Accept()
        if err != nil {
            log.Fatal("Accept error:", err)
        }
        go rpc.Se
```

```
    }
}
```

在新的RPC服务端实现中，我们用RegisterHelloService函数来注册函数，这样不仅可以避免命名服务名称的工作，同时也保证了传入的服务对象满足了RPC接口的定义。最后我们新的服务改为支持多个TCP链接，然后为每个TCP链接提供RPC服务。

跨语言的RPC

标准库的RPC默认采用Go语言特有的gob编码，因此从其它语言调用Go语言实现的RPC服务将比较困难。在互联网的微服务时代，每个RPC以及服务的使用者都可能采用不同的编程语言，因此跨语言是互联网时代RPC的一个首要条件。得益于RPC的框架设计，Go语言的RPC其实也是很容易实现跨语言支持的。

Go语言的RPC框架有两个比较有特色的设计：一个是RPC数据打包时可以通过插件实现自定义的编码和解码；另一个是RPC建立在抽象的io.ReadWriteCloser接口之上的，我们可以将RPC架设在不同的通讯协议之上。这里我们将尝试通过官方自带的net/rpc/jsonrpc扩展实现一个跨语言的RPC。

基于json编码实现RPC服务

```
func main() {
    rpc.RegisterName("HelloService", new(HelloService))
    listener, err := net.Listen("tcp", ":1234")
    if err != nil {
        log.Fatal("ListenTCP error:", err)
    }
    for {
        conn, err := listener.Accept()
        if err != nil {
            log.Fatal("Accept error:", err)
        }
        go rpc.ServeCodec(jsonrpc.NewServerCodec(conn))
    }
}
```

代码中最大的变化是用rpc.ServeCodec函数替代了rpc.ServeConn函数，传入的参数是针对服务端的json编解码器。

然后是实现json版本的客户端：

```
func main() {
    conn, err := net.Dial("tcp", "localhost:1234")
    if err != nil {
        log.Fatal("net.Dial:", err)
    }

    client := rpc.NewClientWithCodec(jsonrpc.NewClientCodec(conn))

    var reply string
    err = client.Call("HelloService.Hello", "hello", &reply)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println(reply)
}
```

先手工调用net.Dial函数建立TCP链接，然后基于该链接建立针对客户端的json编解码器。

在确保客户端可以正常调用RPC服务的方法之后，我们用一个普通的TCP服务代替Go语言版本的RPC服务，这样可以查看客户端调用时发送的数据格式。比如通过nc命令nc -l 1234在同样的端口启动一个TCP服务。然后再次执行一次RPC调用将会发现nc输出了以下的信息：

```
{"method":"HelloService.Hello","params":["hello"],"id":0}
```

这是一个json编码的数据，其中method部分对应要调用的rpc服务和方法组合成的名字，params部分的第一个元素为参数，id是由调用端维护的一个唯一的调用编号。

请求的json数据对象在内部对应两个结构体：客户端是clientRequest，服务端是serverRequest。clientRequest和serverRequest结构体的内容基本是一致的：

```
type clientRequest struct {
    Method string         `json:"method"`
    Params [1]interface{} `json:"params"`
    Id     uint64         `json:"id"`
}

type serverRequest struct {
    Method string           `json:"method"`
    Params *json.RawMessage `json:"params"`
    Id     *json.RawMessage `json:"id"`
}
```

在获取到RPC调用对应的json数据后，我们可以通过直接向架设了RPC服务的TCP服务器发送json数据模拟RPC方法调用：

```
$ echo -e '{"method":"HelloService.Hello","params":["hello"],"id":1}' | nc loca
```

返回的结果也是一个json格式的数据：

```
{"id":1,"result":"hello:hello","error":null}
```

其中id对应输入的id参数，result为返回的结果，error部分在出问题时表示错误信息。对于顺序调用来说，id不是必须的。但是Go语言的RPC框架支持异步调用，当返回结果的顺序和调用的顺序不一致时，可以通过id来识别对应的调用。

返回的json数据也是对应内部的两个结构体：客户端是clientResponse，服务端是serverResponse。两个结构体的内容同样也是类似的：

```
type clientResponse struct {
    Id     uint64           `json:"id"`
    Result *json.RawMessage `json:"result"`
    Error  interface{}      `json:"error"`
}

type serverResponse struct {
    Id     *json.RawMessage `json:"id"`
    Result interface{}      `json:"result"`
    Error  interface{}      `json:"error"`
}
```

因此无论采用何种语言，只要遵循同样的json结构，以同样的流程就可以和Go语言编写的RPC服务进行通信。这样我们就实现了跨语言的RPC。

Http上的RPC

Go语言内在的RPC框架已经支持在Http协议上提供RPC服务。但是框架的http服务同样采用了内置的gob协议，并且没有提供采用其它协议的接口，因此从其它语言依然无法访问的。在前面的例子中，我们已经实现了在TCP协议之上运行jsonrpc服务，并且通过nc命令行工具成功实现了RPC方法调用。现在我们尝试在http协议上提供jsonrpc服务。

新的RPC服务其实是一个类似REST规范的接口，接收请求并采用相应处理流程：

```
func main() {
    rpc.RegisterName("HelloService", new(HelloService))

    http.HandleFunc("/jsonrpc", func(w http.ResponseWriter, r *http.Request) {
        var conn io.ReadWriteCloser = struct {
            io.Writer
            io.ReadCloser
        }{
            ReadCloser: r.Body,
            Writer:     w,
        }
 rpc.ServeRequest(jsonrpc.NewServerCodec(conn))
    })
    http.ListenAndServe(":1234", nil)
}
```

RPC的服务架设在“/jsonrpc”路径，在处理函数中基于http.ResponseWriter和http.Request类型的参数构造一个io.ReadWriteCloser类型的conn通道。然后基于conn构建针对服务端的json编码解码器。最后通过rpc.ServeRequest函数为每次请求处理一次RPC方法调用。

模拟一次RPC调用的过程就是向该链接发送一个json字符串：

```
$ curl localhost:1234/jsonrpc -X POST \
    --data '{"method":"HelloService.Hello","params":["hello"],"id":0}'
```

返回的结果依然是json字符串：

```
{"id":0,"result":"hello:hello","error":null}
```

这样就可以很方便地从不同语言中访问RPC服务了。

发布于 2019-07-11

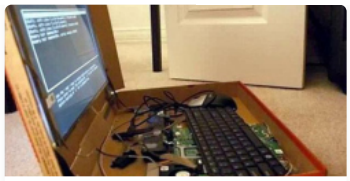
Go 语言　微服务架构　RPC框架

文章被以下专栏收录

Go语言
人生苦短，Let's Go！ 学习交流：2377289333

关注专栏

推荐阅读



2019笔记本电脑选购指南

Jukes同学

在本系列课程前面的内容中，已经学习过Protobuf的相关内容，本节课内容我们尝试将Protobuf与RPC结合在一起使用，通过Protobuf来最终保证RPC的接口规范和安全。再谈Protobuf协议1、创建hello...

Davie 发表于Go语言

Go语言微服务架构实战：第六

Davie 发表于Go语言

Davie

还没有评论

写下你的评论...