

Go语言微服务架构实战：第十七节 Protobuf与RPC



Davie

Java开发 | Android开发 | Go语言编程资深爱好者

关注他

1人赞同了该文章

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.proto文件

```
syntax = "proto3";  
package main;  
message Person {  
    string name = 1;  
    int32 age = 2;  
}
```

说明：* 1、syntax = "proto3";表示采用Proto3的语法。第三版的Protobuf对语言进行了提炼简化，所有成员均采用类似Go语言中的零值初始化（不再支持自定义默认值），因此消息成员也不再需要支持required特性。

- 2、package main;表示使用package指令指明当前是main包（这样可以和Go的包名保持一致，简化例子代码），当然用户也可以针对不同的语言定制对应的包路径和名称。
- 3、message String{};最后message关键字定义一个新的String类型，在最终生成的Go语言代码中对应一个String结构体。String类型中只有一个字符串类型的value成员，该成员编码时用1编号代替名字。

2、编译.proto文件，生成Go代码

proto协议需要安装编译器，安装完以后可以采用编译器进行编译。通过以下命令生成相应的Go代码：

```
$ protoc --go_out=. hello.proto
```

说明：* 1、go_out: go_out参数告知protoc编译器去加载对应的protoc-gen-go工具，然后通过该工具生成代码，生成代码放到当前目录。即go_out用于指定编译后的输出文件的目录。

- hello.proto：代表要编译处理的protobuf文件。此处可以放多个.proto文件，用空格隔开。

Protobuf与RPC结合

基于我们定义的proto文件，重新实现PersonService服务：

```
type PersonService struct{  
}  
  
func (hello *HelloService) Hello(request *person.Person, response *person.Person){  
    response.Name = "姓名: " + request.GetName()  
    response.Age = request.GetAge()  
    return nil  
}
```

▲ 赞同 1 ▼

● 添加评论

➦ 分享

★ 收藏

...



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

    listen, err := net.Listen("tcp", ":1234")
    if err != nil {
        log.Fatal(err)
    }
    conn, err := listen.Accept()
    if err != nil {
        log.Fatal(err)
    }
    rpc.ServeConn(conn)
}
```

RPC实现的原理

Go语言的RPC库最简单的使用方式是通过Client.Call方法进行同步阻塞调用，该方法的实现如下：

```
// Call invokes the named function, waits for it to complete, and returns its error
func (client *Client) Call(serviceMethod string, args interface{}, reply interface{}) error {
    call := <-client.Go(serviceMethod, args, reply, make(chan *Call, 1)).Done
    return call.Error
}
```

首先通过Client.Go方法进行一次异步调用，返回一个表示这次调用的Call结构体。然后等待Call结构体的Done管道返回调用结果。

执行异步调用的Client.Go方法实现如下：

```
func (client *Client) Go(serviceMethod string, args interface{}, reply interface{}) *Call {
    call := new(Call)
    call.ServiceMethod = serviceMethod
    call.Args = args
    call.Reply = reply
    if done == nil {
        done = make(chan *Call, 10) // buffered.
    } else {
        // If caller passes done != nil, it must arrange that
        // done has enough buffer for the number of simultaneous
        // RPCs that will be using that channel. If the channel
        // is totally unbuffered, it's best not to run at all.
        if cap(done) == 0 {
            log.Panic("rpc: done channel is unbuffered")
        }
    }
    call.Done = done
    client.send(call)
    return call
}
```

首先是构造一个表示当前调用的call变量，然后通过client.send将call的完整参数发送到RPC框架。client.send方法调用是线程安全的，因此可以从多个Goroutine同时向同一个RPC链接发送调用指令。

当调用完成或者发生错误时，将调用call.done方法通知完成：

```
func (call *Call)
select {
```

▲ 赞同 1 ▼

● 添加评论

➦ 分享

★ 收藏

...


```
case call.Done <- call:
    // ok
default:
    // We don't want to block here. It is the caller's responsibility to make
    // sure the channel has enough buffer space. See comment in Go().
}
```

从Call.done方法的实现可以得知call.Done管道会将处理后的call返回。

作者：Davie
出处：千锋教育go语言教研部
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

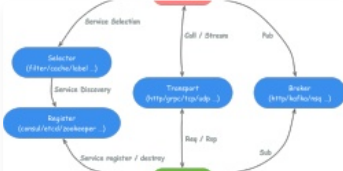
关注专栏

推荐阅读




《暴裂无声》:尸体就在洞里

东山宅 发表于只是电影爱...



微服务架构实践（服务框架）

周小叨



Go语言微服务架构实战：第三节 微服务简介--解决复杂问题

Davie 发表于Go语言

Go语言微服务架构实战：第十七节 Protobuf与RPC

RPC 通信的微服务时，需要一种通信方式（inter-process communication，简称IPC）

Davie

还没有评论

写下你的评论...