三、GRPC

gRPC 是一个高性能、开源和通用的 RPC 框架，面向移动和 HTTP/2 设计。

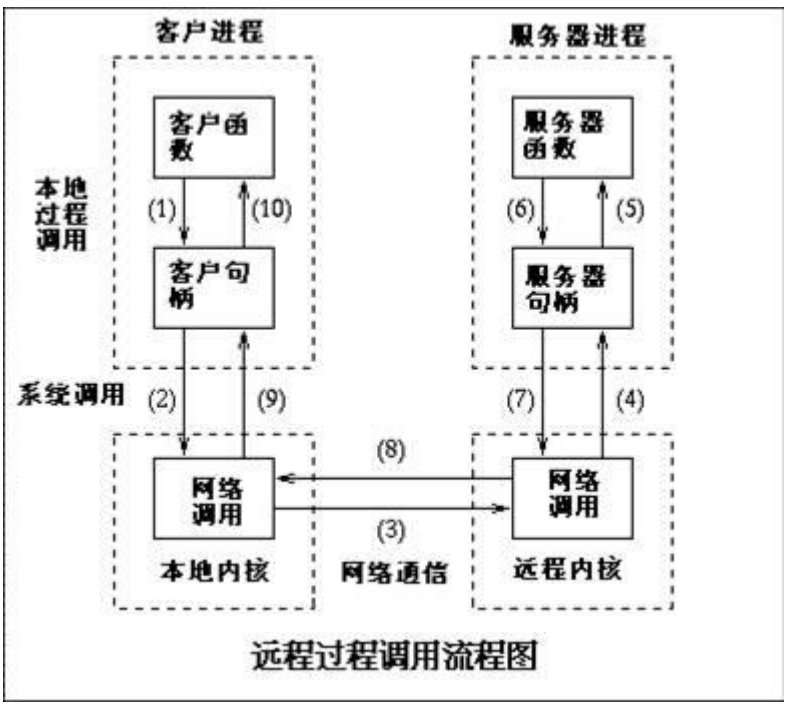
gRPC基于 HTTP/2标准设计，带来诸如双向流、流控、头部压缩、单 TCP连接上的多复用请求等特。这些特性使得其在移动设备上表现更好，更省电和节省空间占用。

RPC

RPC（Remote Procedure Call Protocol）——远程过程调用协议，它是一种通过网络从远程计算机程序上请求服务，而不需要了解底层网络技术的协议。

简单来说，就是跟远程访问或者web请求差不多，都是一个client向远端服务器请求服务返回结果，但是web请求使用的网络协议是http高层协议，而rpc所使用的协议多为TCP，是网络层协议，减少了信息的包装，加快了处理速度。

golang本身有rpc包，可以方便的使用，来构建自己的rpc服务，下边是一个简单是实例，可以加深我们的理解



1.调用客户端句柄；执行传送参数

2.调用本地系统内核发送网络消息

3.消息传送到远程主机

4.服务器句柄得到消息并取得参数

5.执行远程过程

6.执行的过程将结果返回服务器句柄

7.服务器句柄返回结果，调用远程系统内核

8.消息传回本地主机

9.客户句柄由内核接收消息

10.客户接收句柄返回的数据

服务端

```
package main

import (
    "net/http"
    "net/rpc"
    "net"
    "github.com/astaxie/beego"

    "io"
)

//- 方法是导出的
//- 方法有两个参数，都是导出类型或内建类型
//- 方法的第二个参数是指针
//- 方法只有一个error接口类型的返回值
//
//func (t *T) MethodName(argType T1, replyType *T2) error

type Panda int;

func (this *Panda)Getinfo(argType int, replyType *int) error {

    beego.Info(argType)
    *replyType  =1 +argType

    return nil
}

func main() {

    //注册1个页面请求
    http.HandleFunc("/panda",pandatext)
```

```
33
34      //new 一个对象
35      pd :=new(Panda)
36      //注册服务
37      //Register在默认服务中注册并公布 接收服务 pd对象 的方法
38      rpc.Register(pd)
39
40      rpc.HandleHTTP()
41      //建立网络监听
42      ln , err :=net.Listen("tcp","127.0.0.1:10086")
43      if err != nil{
44          beego.Info("网络连接失败")
45      }
46
47      beego.Info("正在监听10086")
48      //service接受侦听器l上传入的HTTP连接,
49      http.Serve(ln,nil)
50
51  }
52  //用来现实网页的web函数
53  func pandatext(w http.ResponseWriter, r *http.Request) {
54      io.WriteString(w,"panda")
55  }
```

客户端

```
1   package main
2
3   import (
4       "net/rpc"
5       "github.com/astaxie/beego"
6   )
7
8   func main() {
9       //rpc的与服务端建立网络连接
10      cli,err := rpc.DialHTTP("tcp","127.0.0.1:10086")
11      if err !=nil {
12          beego.Info("网络连接失败")
13      }
14
15      var val int
16      //远程调用函数（被调用的方法，传入的参数 ，返回的参数）
17      err =cli.Call("Panda.Getinfo",123,&val)
18      if err!=nil{
19          beego.Info("打call失败")
20      }
21      beego.Info("返回结果",val)
22
23  }
24
```

GRPC是什么?

在 gRPC里客户端应用可以像调用本地对象一样直接调用另一台不同的机器上服务端应用的方法，使得您能够更容易地创建分布式应用和服务。与许多 RPC系统类似， gRPC也是基于以下理念：

定义一个服务，指定其能够被远程调用的方法（包含参数和返回类型）。

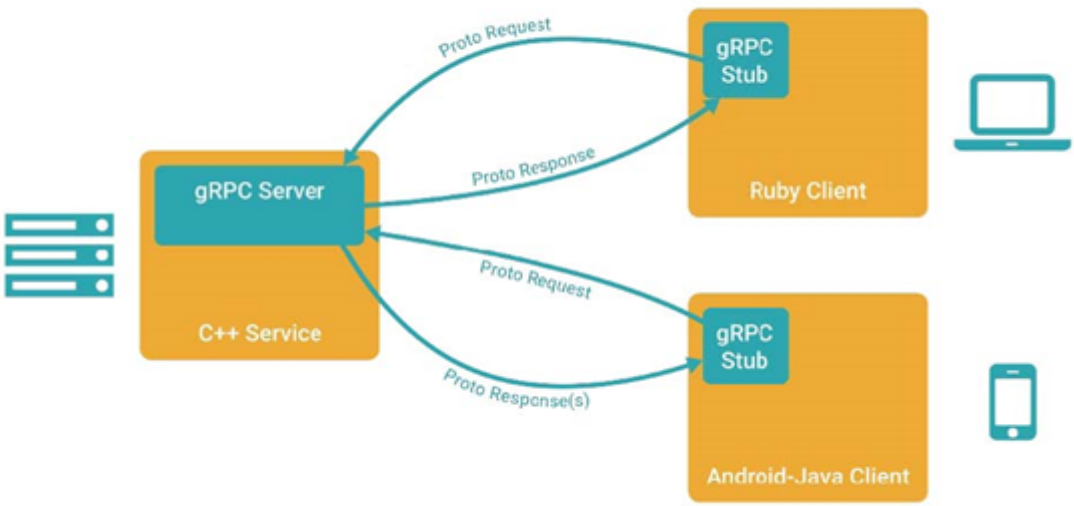
在服务端实现这个接口，并运行一个 gRPC服务器来处理客户端调用。

在客户端拥有一个存根能够像服务端一样的方法。 gRPC客户端和服务端可以在多种环境中运行和交互 -从 google 内部的服务器到你自己的笔记本，并且可以用任何 gRPC支持的语言 来编写。

所以，你可以很容易地用 Java创建一个 gRPC服务端，用 Go、 Python、Ruby来创建客户端。此外， Google最新 API将有 gRPC版本的接口，使你很容易地将 Google的功能集成到你的应用里。

GRPC使用 protocol buffers

```
1 gRPC默认使用protoBuf，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）。正如你将在下方例子里所看到的，你用 proto files创建 gRPC服务，用 protoBuf消息类型来定义方
  法参数和返回类型。你可以在 Protocol Buffers文档找到更多关于 protoBuf的资料。
2 虽然你可以使用 proto2 （当前默认的 protocol buffers版本 ），我们通常建议你在 gRPC里使用 proto3，
  因为这样你可以使用 gRPC支持全部范围的的语言，并且能避免 proto2客户端与 proto3服务端交互时出现的兼
  容性问题，反之亦然。
```



你好 gRPC

现在你已经对 gRPC有所了解，了解其工作机制最简单的方法是看一个简单的例子。 Hello World将带领你创建一个简单的客户端 —— 服务端应用，向你展示：

```
通过一个protoBuf模式，定义一个简单的带有 Hello World方法的 RPC服务。

用你最喜欢的语言（如果可用的话）来创建一个实现了这个接口的服务端。

用你最喜欢的（或者其他你愿意的）语言来访问你的服务端。
```

这个例子完整的代码在我们 GitHub源码库的 examples目录下。我们使用 Git版本系统来进行源码管理，但是除了如何安装和运行一些 Git命令外，你没必要知道其他关于 Git的任何事情。需要注意的是，并不是所有 gRPC支持的语言都可以编写我们例子的服务端代码，比如 PHP和 Objective-C仅支持创建客户端。比起针对于特定语言的复杂教程，这更像是一个介绍性的例子。你可以在本站找到更有深度的教程， gRPC支持的语言的参考文档很快就会全部开放。

环境搭建

```
#将x.zip 解压到 $GOPATH/src/golang.org/x 目录下
$ unzip x.zip -d /GOPATH/src/golang.org/x
#-d 是指定解压目录地址
#/home/itcast/go/src/golang.org
#文件名为x

#将google.golang.org.zip 解压到 $GOPATH/src/google.golang.org 目录下

```

启动服务端

```
$ cd $GOPATH/src/google.golang.org/grpc/examples/helloworld/greeter_server
$ go run main.go
```

启动客户端

```
$ cd $GOPATH/src/google.golang.org/grpc/examples/helloworld/greeter_client
$ go run main.go
```

客户端代码介绍

```
package main
```

```go

import (
    "log"
    "os"

    "golang.org/x/net/context"
    "google.golang.org/grpc"
    pb "google.golang.org/grpc/examples/helloworld/helloworld"
    //这是引用编译好的protobuf
)

const (
    address     = "localhost:50051"
    defaultName = "world"
)

func main() {
    // 建立到服务器的连接。
    conn, err := grpc.Dial(address, grpc.WithInsecure())
    if err != nil {
        log.Fatalf("did not connect: %v", err)
    }
    //延迟关闭连接
    defer conn.Close()
    //调用protobuf的函数创建客户端连接句柄
    c := pb.NewGreeterClient(conn)

    // 联系服务器并打印它的响应。
    name := defaultName
    if len(os.Args) > 1 {
        name = os.Args[1]
    }
    //调用protobuf的sayhello函数
    r, err := c.SayHello(context.Background(), &pb.HelloRequest{Name: name})
    if err != nil {
        log.Fatalf("could not greet: %v", err)
    }
    //打印结果
    log.Printf("Greeting: %s", r.Message)
}
```

服务端代码介绍

```go
package main

import (
    "log"
    "net"
    "golang.org/x/net/context"
    "google.golang.org/grpc"
    pb "google.golang.org/grpc/examples/helloworld/helloworld"
```

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

const (
    port = ":50051"
)

// 服务器用于实现helloworld.GreeterServer。
type server struct{}

// SayHello实现helloworld.GreeterServer
func (s *server) SayHello(ctx context.Context, in *pb.HelloRequest)
(*pb.HelloReply, error) {
    return &pb.HelloReply{Message: "Hello " + in.Name}, nil
}

func main() {
    //监听
    lis, err := net.Listen("tcp", port)
    if err != nil {
        log.Fatalf("failed to listen: %v", err)
    }
    //new服务对象
    s := grpc.NewServer()
    //注册服务
    pb.RegisterGreeterServer(s, &server{})
    // 在gRPC服务器上注册反射服务。
    reflection.Register(s)
    if err := s.Serve(lis); err != nil {
        log.Fatalf("failed to serve: %v", err)
    }
}
```

go语言实现GRPC远程调用

protobuf协议定义

创建一个 protobuf package,如： my_rpc_proto;

在$GOPATH/src/下创建 /my_grpc_proto/文件夹

里面创建 protobuf协议文件 helloServer.proto

```
#到工作目录
$ CD $GOPATH/src/
#创建目录
$ mkdir grpc/myproto
#进入目录
$ cd  grpc/myproto
#创建proto文件
$ vim helloServer.proto
```

文件内容

```
syntax = "proto3";

package my_grpc_proto;

service HelloServer{
//    创建第一个接口
    rpc SayHello(HelloRequest)returns(HelloReplay){}
//    创建第二个接口
    rpc GetHelloMsg(HelloRequest)returns(HelloMessage){}
}

message HelloRequest{
    string name = 1 ;
}
message HelloReplay{
    string message = 1;
}

message HelloMessage{
    string msg = 1;
}
```

在当前文件下，编译 helloServer.proto文件

```
$ protoc --go_out=./ *.proto #不加grpc插件
$ protoc --go_out=plugins=grpc:./ *.proto #添加grpc插件
#对比发现内容增加
#得到 helloServer.pb.go文件
```

gRPC-Server编写

```
package main

import (
    "net"
    "fmt"
    "google.golang.org/grpc"
    pt "demo/grpc/proto"
```

```
    "context"
)

const (
    post  = "127.0.0.1:18881"
)
//对象要和proto内定义的服务一样
type server struct{}

//实现RPC SayHello 接口
func(this *server)SayHello(ctx context.Context,in *pt.HelloRequest)(*pt.HelloReplay
, error){
    return  &pt.HelloReplay{Message:"hello"+in.Name},nil
}
//实现RPC GetHelloMsg 接口
func (this *server) GetHelloMsg(ctx context.Context, in *pt.HelloRequest)
(*pt.HelloMessage, error) {
    return &pt.HelloMessage{Msg: "this is from server HAHA!"}, nil
}

func main() {
    //监听网络
    ln ,err :=net.Listen("tcp",post)
    if err!=nil {
        fmt.Println("网络异常",err)
    }

//  创建一个grpc的句柄
    srv:= grpc.NewServer()
    //将server结构体注册到 grpc服务中
    pt.RegisterHelloServerServer(srv,&server{})

    //监听grpc服务
    err= srv.Serve(ln)
    if err!=nil {
        fmt.Println("网络启动异常",err)
    }

}
```

gRPC-Client编写

```
package main

import (
    "google.golang.org/grpc"
    pt "demo/grpc/proto"
    "fmt"
    "context"
```

```
)

const (
    post  = "127.0.0.1:18881"
)

func main() {

//  客户端连接服务器
    conn,err:=grpc.Dial(post,grpc.WithInsecure())
    if err!=nil {
        fmt.Println("连接服务器失败",err)
    }

    defer conn.Close()

    //获得grpc句柄
    c:=pt.NewHelloServerClient(conn)

    //  远程调用 SayHello接口

    //远程调用 SayHello接口
    r1, err := c.SayHello(context.Background(), &pt.HelloRequest{Name: "panda"})
    if err != nil {
        fmt.Println("cloud not get Hello server ..", err)
        return
    }
    fmt.Println("HelloServer resp: ", r1.Message)

    //远程调用 GetHelloMsg接口
    r2, err := c.GetHelloMsg(context.Background(), &pt.HelloRequest{Name: "panda"})
    if err != nil {
        fmt.Println("cloud not get hello msg ..", err)
        return
    }
    fmt.Println("HelloServer resp: ", r2.Msg)

}
```

运行

```
#先运行 server, 后运行 client

#得到以下输出结果
HelloServer resp:  hellopanda
HelloServer resp:  this is from server HAHA!

#如果反之则会报错
```