如何实现服务(微服务)之间的调用的？一般来说最常用的是两种方式: RESTful API和RPC两种服务通讯方式。

RPC vs RESTful

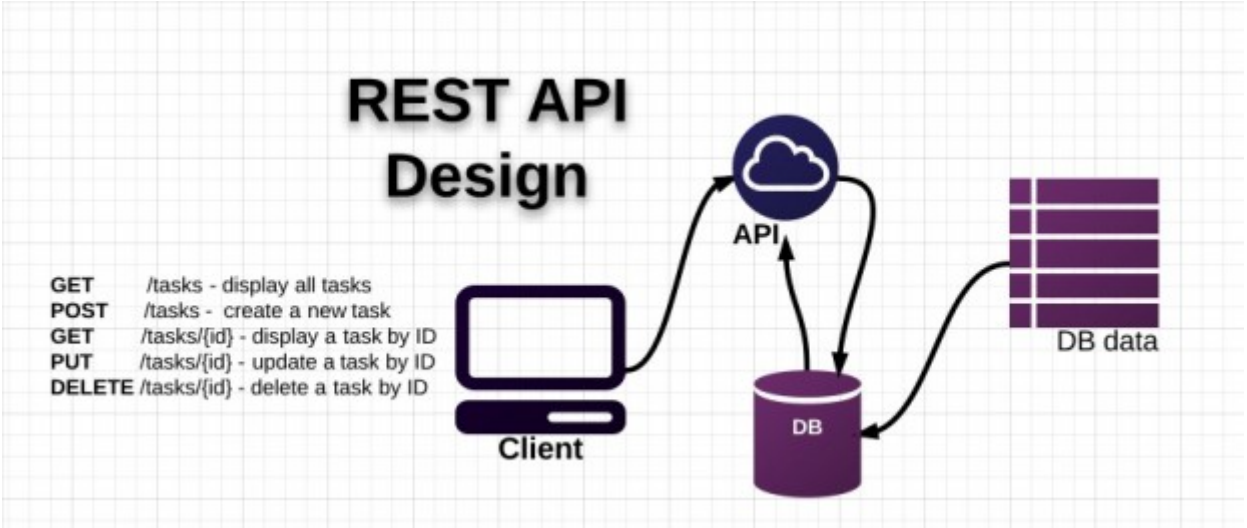
RPC 的消息传输可以通过 TCP、UDP 或者 HTTP等，所以有时候我们称之为 RPC over TCP、 RPC over HTTP。RPC 通过 HTTP 传输消息的时候和RESTful的架构是类似的，但是也有不同。

首先我们比较 RPC over HTTP 和 RESTful。

首先 ，RPC 的客户端和服务器端师紧耦合的，客户端需要知道调用的过程的名字，过程的参数以及它们的类型、顺序等。一旦服务器更改了过程的实现， 客户端的实现很容易出问题。RESTful基于 http的语义操作资源，参数的顺序一般没有关系，也很容易的通过代理转换链接和资源位置，从这一点上来说，RESTful 更灵活。

其次，它们操作的对象不一样。 RPC 操作的是方法和过程，它要操作的是方法对象。 RESTful 操作的是资源(resource)，而不是方法。

第三，RESTful执行的是对资源的操作，增加、查找、修改和删除等,主要是CURD，所以如果你要实现一个特定目的的操作，比如为名字姓张的学生的数学成绩都加上10这样的操作， RESTful的API设计起来就不是那么直观或者有意义。在这种情况下, RPC的实现更有意义，它可以实现一个 `Student.Increment(Name, Score)` 的方法供客户端调用。



我们再来比较一下 RPC over TCP 和 RESTful。 如果我们直接使用socket实现RPC，除了上面的不同外，我们可以获得性能上的优势。

RPC over TCP可以通过长连接减少连接的建立所产生的花费，在调用次数非常巨大的时候(这是目前互联网公司经常遇到的情况,大并发的情况下)，这个花费影响是非常巨大的。 当然 RESTful 也可以通过 keep-alive 实现长连接， 但是它最大的一个问题是它的request-response模型是阻塞的 (http1.0和 http1.1, http 2.0没这个问题)， 发送一个请求后只有等到response返回才能发送第二个请求 (有些http server实现了pipeling的功能，但不是标配)， RPC的实现没有这个限制。在当今用户和资源都是大数据大并发的趋势下，一个大规模的公司不可能使用一个单体程序提供所有的功能，微服务的架构模式越来越多的被应用到产品的设计和开发中， 服务和服务之间的通讯也越发的重要， 所以 RPC 不失是一个解决服务之间通讯的好办法， 本章给大家介绍 Go 语言的 gRPC的开发实践。

proc命令的使用：

protoc --go_out=./pbgo -I pb ./pb/helloworld.proto

--go_out 为输出go代码的输出目录，这里指定的是 ./pbgo / 目录。
随后我们指定了proto文件的位置 ./pb/helloworld.proto 。
执行上述命令，我们就 ./pbgo / 目录下就产生了对应的 go文件。
-I=*IMPORT_PATH*
简单来说，就是如果多个proto文件之间有互相依赖，生成某个proto文件时，需要import其他几个proto文件，这时候就要用-I来指定搜索目录。
如果没有指定 –I 参数，则在当前目录进行搜索。

如proto文件中定义了grpc 服务，则需向下面格式一样指定插件和输出目录（【=grpc:】后的就是输出目录）

protoc --go_out=plugins=grpc:pbgo/greeter -I pb ./pb/helloworld.proto

gprc安装：

go get google.golang.org/grpc
是安装不起的，会报：

package google.golang.org/grpc: unrecognized import path "google.golang.org/grpc"(https fetch: Get https://google.golang.org/grpc?go-get=1: dial tcp 216.239.37.1:443: i/o timeout)
原因是这个代码已经转移到github上面了，但是代码里面的包依赖还是没有修改，还是google.golang.org这种，

所以不能使用go get的方式安装，正确的安装方式：

```
git clone https://github.com/grpc/grpc-go.git $GOPATH/src/google.golang.org/grpc
git clone https://github.com/golang/net.git $GOPATH/src/golang.org/x/net
git clone https://github.com/golang/text.git $GOPATH/src/golang.org/x/text
go get -u github.com/golang/protobuf/{proto,protoc-gen-go}
git clone https://github.com/google/go-genproto.git $GOPATH/src/google.golang.org/genproto
cd $GOPATH/src/
go install google.golang.org/grpc
```

1:编写proto文件，
在proto文件中定义消息类型，并声明服务。
helloword.proto文件

```
syntax = "proto3";
//这个文件定义了一个Greeter服务，它有一个SayHello方法，
//这个方法接收一个Request，返回一个Response。
package greeter;
//服务定义
service Greeter{
	rpc SayHello(HelloRequest) returns (HelloReply){}
}

//request 消息类型
message HelloRequest {
	string name = 1;
```

```
}

message HelloReply{
	string message = 1;
}
```

2:根据proto文件生成对应的go语言文件（helloworld.pb.go）

protoc --go_out=plugins=grpc:pbgo/greeter -I pb ./pb/helloworld.proto

helloworld.proto文件在./pb 目录下

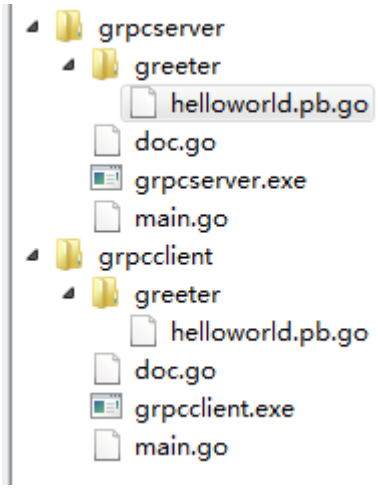
helloworld.pb.go文件在./pbgo/greeter目录下

3:创建服务端

创建一个go commond 项目，

并将将**helloworld.pb.go 文件拷贝到greeter目录下（因为在proto文件中定义了greeter包）**

如下图所示：




```
// grpcserver project main.go
// grpc 服务端
package main

import (
    pb "grpcserver/greeter"
    "log"
    "net"
```

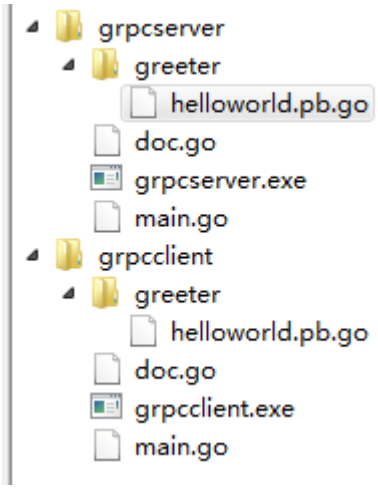
```
    "golang.org/x/net/context"

    "google.golang.org/grpc"
)

//定义需监听的端口
const (
    port = ":50051"
)

//实现GreeterServer接口
type server struct{}

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

func main() {
    //tcp方式监听端口
    lis, err := net.Listen("tcp", port)
    if err != nil {
        log.Fatalf("failed to listen : %v", port)
    }
    s := grpc.NewServer()
    //注册GreeterServer接口的实现来响应客户端的请求
    pb.RegisterGreeterServer(s, &server{})
    s.Serve(lis)
}
```

4：创建客户端

创建一个go commond 项目，

并将将**helloworld.pb.go 文件拷贝到greeter目录下（因为在proto文件中定义了greeter包）**

如下图所示：




```
// grpcclient project main.go
package main

import (
    "flag"
    "fmt"
    pb "grpcclient/greeter"
    "log"
    "sync"
    "time"

    "golang.org/x/net/context"
    "google.golang.org/grpc"
)

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

//是否异步调用
var isAsync bool
```

```
func init() {
    flag.BoolVar(&isAsync, "a", true, "是否异步调用，默认是")
}

//GreeterClient is the client API for Greeter service
//GreeterClient中定义Greeter服务中可以远程调用的方法
func invoke(c pb.GreeterClient, name string) {
    r, err := c.SayHello(context.Background(), &pb.HelloRequest{Name: name})
    if err != nil {
        log.Fatalf("could not greet :%v", err)
    }
    _ = r
}

//同步调用
func syncTest(c pb.GreeterClient, name string) {
    i := 10000
    t := time.Now().UnixNano()
    for ; i > 0; i-- {
        invoke(c, name)
    }
    //计算服务调用耗时，本地测试大概230ms
    fmt.Println("took", (time.Now().UnixNano()-t)/1000000, "ms")
}

func asyncTest(c [20]pb.GreeterClient, name string) {
    var wg sync.WaitGroup
    wg.Add(10000)
    t := time.Now().UnixNano()
    i := 10000
    for ; i > 0; i-- {
        go func() {
            invoke(c[i%20], name)
            wg.Done()
        }()
```

```
    }
    wg.Wait()
    fmt.Println("took", (time.Now().UnixNano()-t)/1000000, "ms")

}

func main() {
    flag.Parse()
    //创建客户端连接
    conn, err := grpc.Dial(address, grpc.WithInsecure())
    if err != nil {
        log.Fatalf("did not connect: %v", err)
    }
    defer conn.Close()
    var c [20]pb.GreeterClient
    name := defaultName

    i := 0
    for ; i < 20; i++ {
        c[i] = pb.NewGreeterClient(conn)
        invoke(c[i], name)
    }

    if isAsync {
        asyncTest(c, name)
    } else {
        syncTest(c[0], name)
    }

}
```

5： 编译后先启动服务端，在运行客户端。