

Golang 微服务教程（一）

📅 2018-05-10 | 👁 4388 | 📄 4,073

原文链接：ewanvalentine.io，翻译已获作者 [Ewan Valentine](#) 授权。
本节对 gRPC 的使用浅尝辄止，更多可参考：[gRPC 中 Client 与 Server 数据交互的 4 种模式](#)

前言

系列概览

《Golang 微服务教程》分为 10 篇，总结微服务开发、测试到部署的完整过程。
本节先介绍微服务的基础概念、术语，再创建我们的第一个微服务 `consignment-service` 的简洁版。在接下来的第 2~10 节文章中，我们会陆续创建以下微服务：

- `consignment-service`（货运服务）
- `inventory-service`（仓库服务）
- `user-service`（用户服务）
- `authentication-service`（认证服务）
- `role-service`（角色服务）
- `vessel-service`（货船服务）

用到的完整技术栈如下：

1	Golang, gRPC, go-micro	// 开发语言及其 RPC 框架
2	Google Cloud, MongoDB	// 云平台与数据存储
3	Docker, Kubernetes, Terraform	// 容器化与集群架构
4	NATS, CircleCI	// 消息系统与持续集成

代码仓库

作者代码：[EwanValentine/shippy](#)，译者的中文注释代码：[wuYin/shippy](#)
每个章节对应仓库的一个分支，比如本文part1 的代码在 [feature/part1](#)

开发环境

笔者的开发环境为 macOS，本文中使用了 make 工具来高效编译，Windows 用户需 [手动安装](#)

```
1  $ go env
2  GOARCH="amd64"    # macOS 环境
3  GOOS="darwin"     # 在第二节使用 Docker 构建 alpine 镜像时需修改为 linux
4  GOPATH="/Users/wuyin/Go"
5  GOROOT="/usr/local/go"
```

准备

掌握 Golang 的基础语法：推荐阅读谢大的 [《Go Web 编程》](#)

安装 [gRPC / protobuf](#)

```
1  go get -u google.golang.org/grpc                                # 安装 g
2  go get -u github.com/golang/protobuf/protoc-gen-go            # 安装 Go 版本的 protobuf
```

微服务

我们要写什么项目？

我们要搭建一个港口的货物管理平台。本项目以微服务的架构开发，整体简单且概念通用。闲话不多说让我们开始微服务之旅吧。

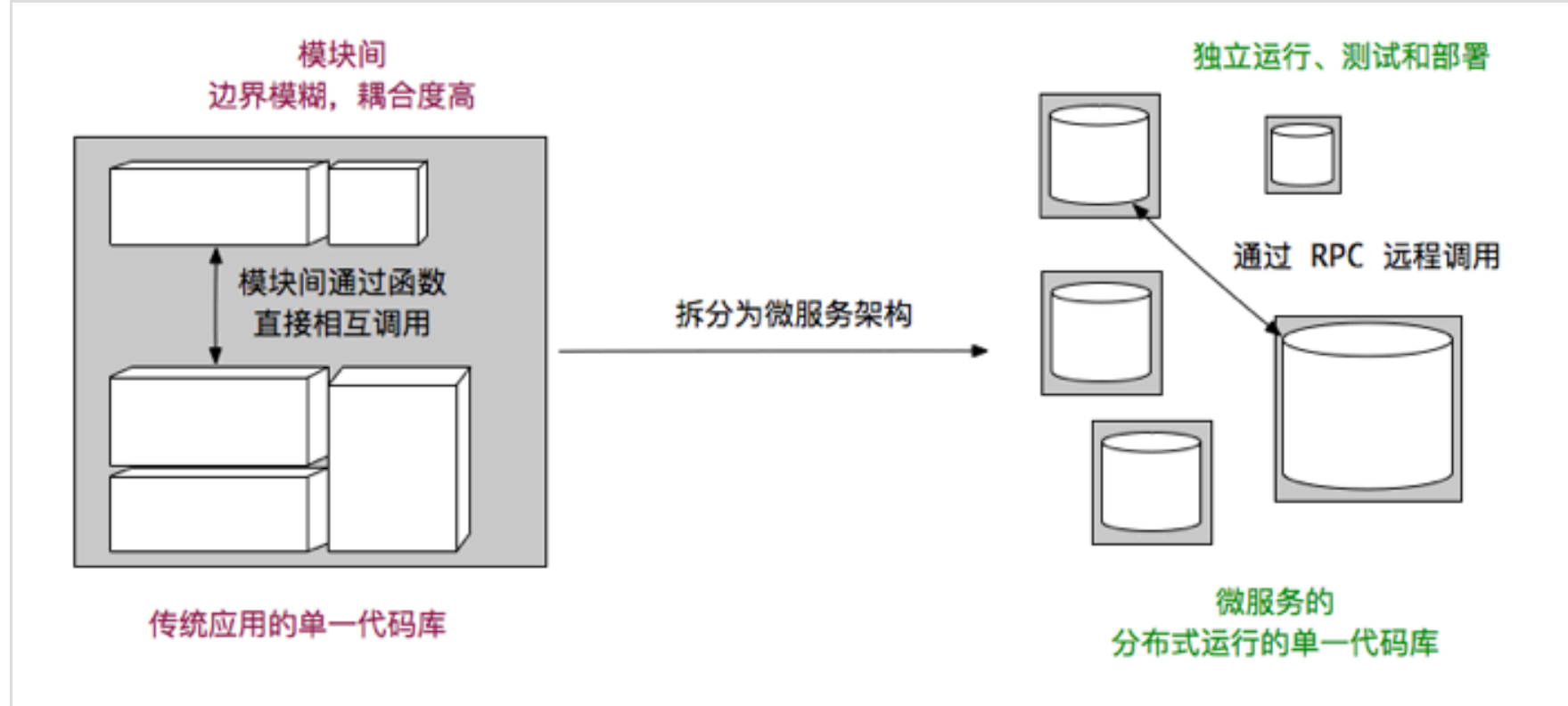
微服务是什么？

在传统的软件开发中，整个应用的代码都组织在一个单一的代码库，一般会有以下拆分代码的形式：

- 按照特征做拆分：如 MVC 模式
- 按照功能做拆分：在更大的项目中可能会将代码封装在处理不同业务的包中，包内部可能会再做拆分

不管怎么拆分，最终二者的代码都会集中在一个库中进行开发和管理，可参考：[谷歌的单一代码库管理](#)

微服务是上述第二种拆分方式的拓展，按功能将代码拆分成几个包，都是可独立运行的单一代码库。区别如下：



微服务有哪些优势？

降低复杂性

将整个应用的代码按功能对应拆分为小且独立的微服务代码库，这不禁让人联想到 Unix 哲学：Do One Thing and Do It Well，在传统单一代码库的应用中，模块之间是紧耦合且边界模糊的，随着产品不断迭代，代码的开发和维护将变得更为复杂，潜在的 bug 和漏洞也会越来越多。

提高扩展性

在项目开发中，可能有一部分代码会在多个模块中频繁的被用到，这种复用性很高的模块常常会抽离出来作为公共代码库使用，比如验证模块，当它要扩展功能（添加短信验证码登录等）时，单一代码库的规模只增不减，整个应用还需重新部署。在微服务架构中，验证模块可作为单个服务独立出来，能独立运行、测试和部署。

遵循微服务拆分代码的理念，能大大降低模块间的耦合性，横向扩展也会容易许多，正适合当下云计算的高性能、高可用和分布式的开发环境。

Nginx 有一系列文章来探讨微服务的许多概念，可 [点此阅读](#)

使用 Golang 的好处？

微服务是一种架构理念而不是具体的框架项目，许多编程语言都可以实现，但有的语言对微服务开发具备天生的优势，Golang 便是其中之一

Golang 本身十分轻量级，运行效率极高，同时对并发编程有着原生的支持，从而能更好的利用多核处理器。内置 `net` 标准库对网络开发的支持也十分完善。可参考谢大的短文：[Go 语言的优势](#)

此外，Golang 社区有一个很棒的开源微服务框架 [go-mirco](#)，我们在下一节会用到。

Protobuf 与 gRPC

在传统应用的单一代码库中，各模块间可直接相互调用函数。但在微服务架构中，由于每个服务对应的代码库是独立运行的，无法直接调用，彼此间的通信就是个大问题，解决方案有 2 个：

JSON 或 XML 协议的 API

微服务之间可使用基于 HTTP 的 JSON 或 XML 协议进行通信：服务 A 与服务 B 进行通信前，A 必须把要传递的数据 encode 成 JSON / XML 格式，再以字符串的形式传递给 B，B 接收到数据需要 decode 后才能在代码中使用：

- 优点：数据易读，使用便捷，是与浏览器交互必选的协议
- 缺点：在数据量大的情况下 encode、decode 的开销随之变大，多余的字段信息导致传输成本更高

RPC 协议的 API

下边的 JSON 数据就使用 `description`、`weight` 等元数据来描述数据本身的意义，在 Browser / Server 架构中用得很多，以方便浏览器解析：

```
1  {
2    "description": "This is a test consignment",
3    "weight": 550,
4    "containers": [
5      {
6        "customer_id": "cust001",
7        "user_id": "user001",
8        "origin": "Manchester, United Kingdom"
9      }
10   ],
11   "vessel_id": "vessel001"
12 }
```

但在两个微服务之间通信时，若彼此约定好传输数据的格式，可直接使用二进制数据流进行通信，不再需要笨重冗余的元数据。

gRPC 简介

gRPC 是谷歌开源的轻量级 RPC 通信框架，其中的通信协议基于二进制数据流，使得 gRPC 具有优异的性能。gRPC 支持 HTTP 2.0 协议，使用二进制帧进行数据传输，还可以为通信双方建立持续的双向数据流。可参考：[Google HTTP/2 简介](#)

protobuf 作为通信协议

两个微服务之间通过基于 HTTP 2.0 二进制数据帧通信，那么如何约定二进制数据的格式呢？答案是使用 gRPC 内置的 protobuf 协议，其 DSL 语法可清晰定义服务间通信的数据结构。可参考：[gRPC Go: Beyond the basics](#)

consignment-service 微服务开发

经过上边必要的概念解释，现在让我们开始开发我们的第一个微服务：**consignment-service**

项目结构

假设本项目名为 **shippy**，你需要：

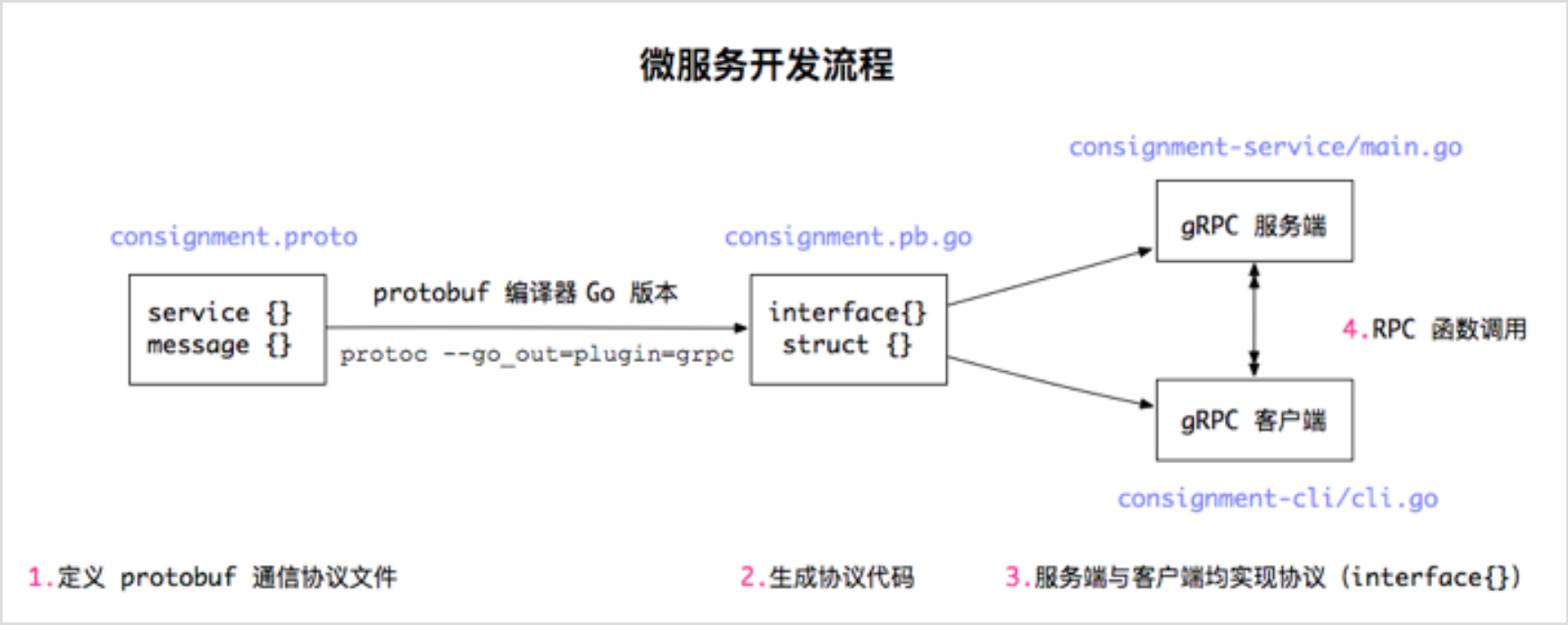
- 在 `$GOPATH` 的 `src` 目录下新建 `shippy` 项目目录
- 在项目目录下新建文件 `consignment-service/proto/consignment/consignment.proto`

为便于教学，我会把本项目的所有微服务的代码统一放在 `shippy` 目录下，这种项目结构被称为“**mono-repo**”，读者也可以按照“**multi-repo**”将各个微服务拆为独立的项目。更多参考 [REPO 风格之争：MONO VS MULTI](#)

现在你的项目结构应该如下：

```
1  $GOPATH/src
2      └─ shippy
3          └─ consignment-service
4              └─ proto
5                  └─ consignment
6                      └─ consignment.proto
```

开发流程



定义 `protobuf` 通信协议文件

```
1  // shipper/consignment-service/proto/consignment/consignment.proto
2
3  syntax = "proto3";
4  package go.micro.srv.consignment;
```

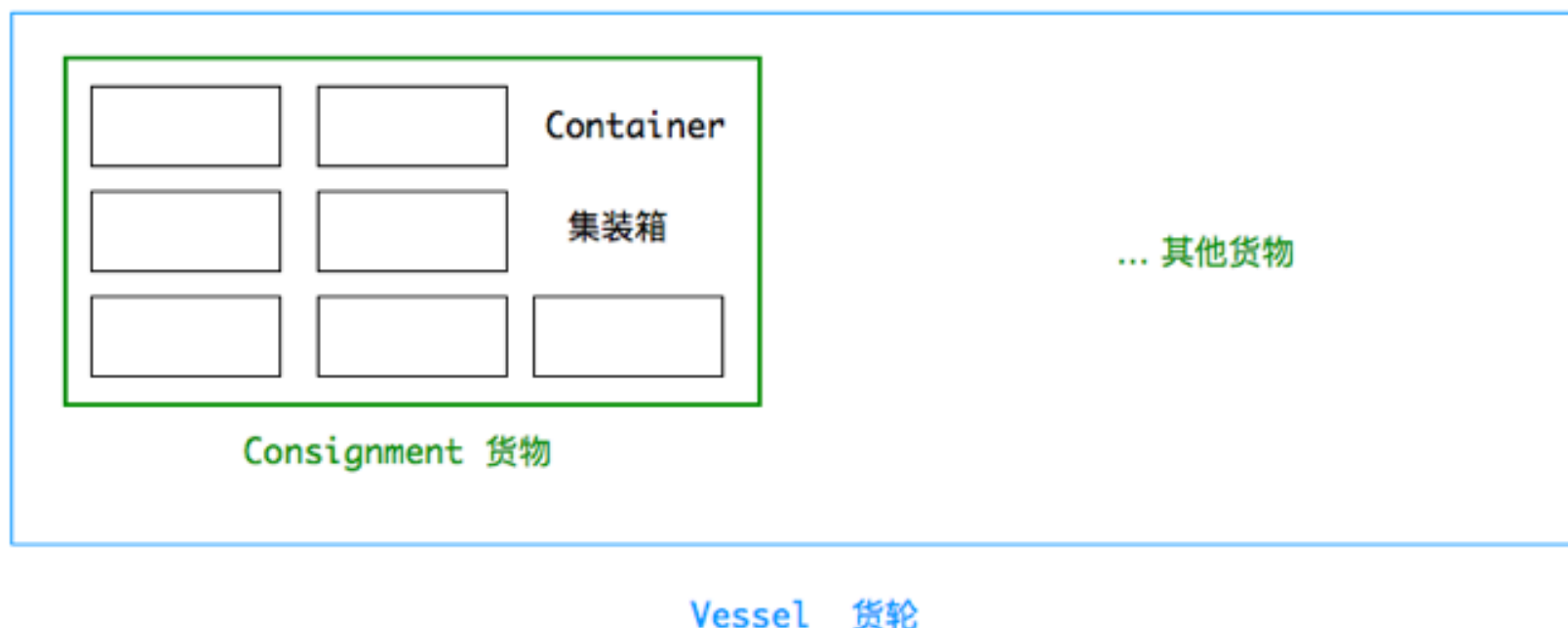
```

5
6 // 货轮微服务
7 service ShippingService {
8     // 托运一批货物
9     rpc CreateConsignment (Consignment) returns (Response) {
10     }
11 }
12
13 // 货轮承运的一批货物
14 message Consignment {
15     string id = 1; // 货物编号
16     string description = 2; // 货物描述
17     int32 weight = 3; // 货物重量
18     repeated Container containers = 4; // 这批货有哪些集装箱
19     string vessel_id = 5; // 承运的货轮
20 }
21
22 // 单个集装箱
23 message Container {
24     string id = 1; // 集装箱编号
25     string customer_id = 2; // 集装箱所属客户的编号
26     string origin = 3; // 出发地
27     string user_id = 4; // 集装箱所属用户的编号
28 }
29
30 // 托运结果
31 message Response {
32     bool created = 1; // 托运成功
33     Consignment consignment = 2; // 新托运的货物
34 }

```

语法参考： [Protobuf doc](#)

ShippingService 承运服务



生成协议代码

protoc 编译器使用 grpc 插件编译 .proto 文件

为避免重复的在终端执行编译、运行命令，本项目使用 make 工具，新建 `consignment-service/Makefile`

```
1 build:
2 # 一定要注意 Makefile 中的缩进，否则 make build 可能报错 Nothing to be done for build
3 # protoc 命令前边是一个 Tab，不是四个或八个空格
4     protoc -I. --go_out=plugins=grpc:$(GOPATH)/src/shippy/consignment-service
```

执行 `make build`，会在 `proto/consignment` 目录下生成 `consignment.pb.go`

consignment.proto 与 consignment.pb.go 的对应关系

service：定义了微服务 ShippingService 要暴露为外界调用的函数：`CreateConsignment`，由 protobuf 编译器的 grpc 插件处理后生成 **interface**

```
1 type ShippingServiceClient interface {
2     // 托运一批货物
3     CreateConsignment(ctx context.Context, in *Consignment, opts ...grpc.CallOptions) (*Consignment, error)
4 }
```

message：定义了通信的数据格式，由 protobuf 编译器处理后生成 **struct**

```
1 type Consignment struct {
2     Id          string          `protobuf:"bytes,1,opt,name=id" json:"id,omitempty"`
3     Description string          `protobuf:"bytes,2,opt,name=description" json:"description"`
4     Weight      int32           `protobuf:"varint,3,opt,name=weight" json:"weight"`
5     Containers  []*Container    `protobuf:"bytes,4,rep,name=containers" json:"containers"`
6     // ...
7 }
```

实现服务端

服务端需实现 `ShippingServiceClient` 接口，创建 `consignment-service/main.go`

```
1 package main
2
3 import (
```

```
4 // 导如 protoc 自动生成的包
5     pb "shippy/consignment-service/proto/consignment"
6     "context"
7     "net"
8     "log"
9     "google.golang.org/grpc"
10 )
11
12 const (
13     PORT = ":50051"
14 )
15
16 //
17 // 仓库接口
18 //
19 type IRepository interface {
20     Create(consignment *pb.Consignment) (*pb.Consignment, error) // 存放新货
21 }
22
23 //
24 // 我们存放多批货物的仓库, 实现了 IRepository 接口
25 //
26 type Repository struct {
27     consignments []*pb.Consignment
28 }
29
30 func (repo *Repository) Create(consignment *pb.Consignment) (*pb.Consignment, error) {
31     repo.consignments = append(repo.consignments, consignment)
32     return consignment, nil
33 }
34
35 func (repo *Repository) GetAll() []*pb.Consignment {
36     return repo.consignments
37 }
38
39 //
40 // 定义微服务
41 //
42 type service struct {
43     repo Repository
44 }
45
46 //
47 // service 实现 consignment.pb.go 中的 ShippingServiceServer 接口
48 // 使 service 作为 gRPC 的服务端
49 //
50 // 托运新的货物
51 func (s *service) CreateConsignment(ctx context.Context, req *pb.Consignment)
```

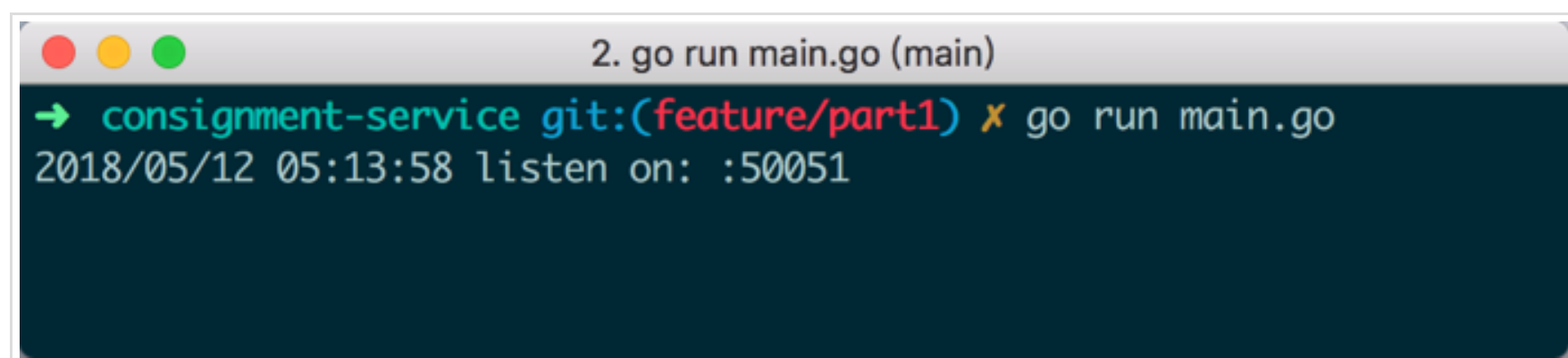


```

52 // 接收承运的货物
53 consignment, err := s.repo.Create(req)
54 if err != nil {
55     return nil, err
56 }
57 resp := &pb.Response{Created: true, Consignment: consignment}
58 return resp, nil
59 }
60
61 func main() {
62     listener, err := net.Listen("tcp", PORT)
63     if err != nil {
64         log.Fatalf("failed to listen: %v", err)
65     }
66     log.Printf("listen on: %s\n", PORT)
67
68     server := grpc.NewServer()
69     repo := Repository{}
70
71     // 向 rRPC 服务器注册微服务
72     // 此时会把我们自己实现的微服务 service 与协议中的 ShippingServiceServer 绑定
73     pb.RegisterShippingServiceServer(server, &service{repo})
74
75     if err := server.Serve(listener); err != nil {
76         log.Fatalf("failed to serve: %v", err)
77     }
78 }

```

上边的代码实现了 consignment-service 微服务所需要的方法，并建立了一个 gRPC 服务器监听 50051 端口。如果你此时运行 `go run main.go`，将成功启动服务端：



```

2. go run main.go (main)
→ consignment-service git:(feature/part1) x go run main.go
2018/05/12 05:13:58 listen on: :50051

```

实现客户端

我们将要托运的货物信息放到 `consignment-cli/consignment.json`：

```

1 {
2     "description": "This is a test consignment",
3     "weight": 550,

```

```

4     "containers": [
5         {
6             "customer_id": "cust001",
7             "user_id": "user001",
8             "origin": "Manchester, United Kingdom"
9         }
10    ],
11    "vessel_id": "vessel001"
12 }

```

客户端会读取这个 JSON 文件并将该货物托运。在项目目录下新建文件：`consingment-cli/cli.go`

```

1  package main
2
3  import (
4      pb "shippy/consingment-service/proto/consingment"
5      "io/ioutil"
6      "encoding/json"
7      "errors"
8      "google.golang.org/grpc"
9      "log"
10     "os"
11     "context"
12 )
13
14 const (
15     ADDRESS          = "localhost:50051"
16     DEFAULT_INFO_FILE = "consingment.json"
17 )
18
19 // 读取 consingment.json 中记录的货物信息
20 func parseFile(fileName string) (*pb.Consingment, error) {
21     data, err := ioutil.ReadFile(fileName)
22     if err != nil {
23         return nil, err
24     }
25     var consingment *pb.Consingment
26     err = json.Unmarshal(data, &consingment)
27     if err != nil {
28         return nil, errors.New("consingment.json file content error")
29     }
30     return consingment, nil
31 }
32
33 func main() {
34     // 连接到 gRPC 服务器
35     conn, err := grpc.Dial(ADDRESS, grpc.WithInsecure())

```

```

36     if err != nil {
37         log.Fatalf("connect error: %v", err)
38     }
39     defer conn.Close()
40
41     // 初始化 gRPC 客户端
42     client := pb.NewShippingServiceClient(conn)
43
44     // 在命令行中指定新的货物信息 json 文件
45     infoFile := DEFAULT_INFO_FILE
46     if len(os.Args) > 1 {
47         infoFile = os.Args[1]
48     }
49
50     // 解析货物信息
51     consignment, err := parseFile(infoFile)
52     if err != nil {
53         log.Fatalf("parse info file error: %v", err)
54     }
55
56     // 调用 RPC
57     // 将货物存储到我们自己的仓库里
58     resp, err := client.CreateConsignment(context.Background(), consignment)
59     if err != nil {
60         log.Fatalf("create consignment error: %v", err)
61     }
62
63     // 新货物是否托运成功
64     log.Printf("created: %t", resp.Created)
65 }

```

运行 `go run main.go` 后再运行 `go run cli.go` :



The image shows two terminal windows. The top window is titled '2. wuyin@fuwafuwa: ~/Go/src/shipper/consignment-service (zsh)' and shows the command 'consignment-service go_run cli.go' being executed. The bottom window is titled '3. wuyin@fuwafuwa: ~/Go/src/shipper/consignment-cli (zsh)' and shows the command 'consignment-cli _' being executed.

```

2. wuyin@fuwafuwa: ~/Go/src/shipper/consignment-service (zsh)
→ consignment-service go_run cli.go

3. wuyin@fuwafuwa: ~/Go/src/shipper/consignment-cli (zsh)
→ consignment-cli _

```

我们可以新增一个 RPC 查看所有被托运的货物，加入一个 `GetConsignments` 方法，这样，我们就能看到所有存在的 `consignment` 了：

```
1 // shipper/consignment-service/proto/consignment/consignment.proto
2
3 syntax = "proto3";
4
5 package go.micro.srv.consignment;
6
7 // 货轮微服务
8 service ShippingService {
9     // 托运一批货物
10     rpc CreateConsignment (Consignment) returns (Response) {
11     }
12     // 查看托运货物的信息
13     rpc GetConsignments (GetRequest) returns (Response) {
14     }
15 }
16
17 // 货轮承运的一批货物
18 message Consignment {
19     string id = 1; // 货物编号
20     string description = 2; // 货物描述
21     int32 weight = 3; // 货物重量
22     repeated Container containers = 4; // 这批货有哪些集装箱
23     string vessel_id = 5; // 承运的货轮
24 }
25
26 // 单个集装箱
27 message Container {
28     string id = 1; // 集装箱编号
29     string customer_id = 2; // 集装箱所属客户的编号
30     string origin = 3; // 出发地
31     string user_id = 4; // 集装箱所属用户的编号
32 }
33
34 // 托运结果
35 message Response {
36     bool created = 1; // 托运成功
37     Consignment consignment = 2; // 新托运的货物
38     repeated Consignment consignments = 3; // 目前所有托运的货物
39 }
40
41 // 查看货物信息的请求
42 // 客户端想要从服务端请求数据，必须有请求格式，哪怕为空
43 message GetRequest {
44 }
```

现在运行 `make build` 来获得最新编译后的微服务界面。如果此时你运行 `go run main.go`，你会获得一个类似这样的错误信息：

```
2. wuyin@fuwafuwa: ~/Go/src/shippy/consignment-service (zsh)
→ consignment-service git:(feature/part1) ✗ go run main.go
# command-line-arguments
./main.go:70:52: cannot use service literal (type *service) as type go_micro_srv_consignment.ShippingServiceServer
in argument to go_micro_srv_consignment.RegisterShippingServiceServer:
    *service does not implement go_micro_srv_consignment.ShippingServiceServer (missing GetConsignments method)
```

熟悉Go的你肯定知道，你忘记实现一个 `interface` 所需要的方法了。让我们更新 `consignment-service/main.go`：

```
1 package main
2
3 import (
4     pb "shippy/consignment-service/proto/consignment"
5     "context"
6     "net"
7     "log"
8     "google.golang.org/grpc"
9 )
10
11 const (
12     PORT = ":50051"
13 )
14
15 //
16 // 仓库接口
17 //
18 type IRepository interface {
19     Create(consignment *pb.Consignment) (*pb.Consignment, error) // 存放新货
20     GetAll() []*pb.Consignment // 获取仓库
21 }
22
23 //
24 // 我们存放多批货物的仓库，实现了 IRepository 接口
25 //
26 type Repository struct {
27     consignments []*pb.Consignment
28 }
29
30 func (repo *Repository) Create(consignment *pb.Consignment) (*pb.Consignment, error) {
31     repo.consignments = append(repo.consignments, consignment)
32     return consignment, nil
33 }
34
35 func (repo *Repository) GetAll() []*pb.Consignment {
36     return repo.consignments
```

```
37 }
38
39 //
40 // 定义微服务
41 //
42 type service struct {
43     repo Repository
44 }
45
46 //
47 // 实现 consignment.pb.go 中的 ShippingServiceServer 接口
48 // 使 service 作为 gRPC 的服务端
49 //
50 // 托运新的货物
51 func (s *service) CreateConsignment(ctx context.Context, req *pb.Consignment)
52     // 接收承运的货物
53     consignment, err := s.repo.Create(req)
54     if err != nil {
55         return nil, err
56     }
57     resp := &pb.Response{Created: true, Consignment: consignment}
58     return resp, nil
59 }
60
61 // 获取目前所有托运的货物
62 func (s *service) GetConsignments(ctx context.Context, req *pb.GetRequest) (*pb
63     allConsignments := s.repo.GetAll()
64     resp := &pb.Response{Consignments: allConsignments}
65     return resp, nil
66 }
67
68 func main() {
69     listener, err := net.Listen("tcp", PORT)
70     if err != nil {
71         log.Fatalf("failed to listen: %v", err)
72     }
73     log.Printf("listen on: %s\n", PORT)
74
75     server := grpc.NewServer()
76     repo := Repository{}
77     pb.RegisterShippingServiceServer(server, &service{repo})
78
79     if err := server.Serve(listener); err != nil {
80         log.Fatalf("failed to serve: %v", err)
81     }
82 }
```


如果现在使用 `go run main.go`，一切应该正常：

```
2. go run main.go (main)
→ consignment-service git:(feature/part1) ✕ go run main.go
2018/05/12 02:01:17 listen on: :50051
```

最后让我们更新 `consignment-cli/cli.go` 来获得 `consignment` 信息：

```
1 func main() {
2     ...
3
4     // 列出目前所有托运的货物
5     resp, err = client.GetConsignments(context.Background(), &pb.GetRequest{
6     if err != nil {
7         log.Fatalf("failed to list consignments: %v", err)
8     }
9     for _, c := range resp.Consignments {
10        log.Printf("%+v", c)
11    }
12 }
```

此时再运行 `go run cli.go`，你应该能看到所创建的所有 `consignment`，多次运行将看到多个货物被托运：

```
1. go run main.go (main)
→ consignment-service git:(feature/part1) ✕ go run main.go
2018/05/12 05:31:11 listen on: :50051

2. wuyin@fuwafuwa: ~/Go/src/shippy/consignment-cli (zsh)
→ consignment-cli git:(feature/part1) ✕ go run cli.go
2018/05/12 05:31:16 created: true
2018/05/12 05:31:16 description:"This is a test consignment" weight:550 containers:<customer_id:"cust001" origin:"Manchester, United Kingdom" user_id:"user001" > vessel_id:"vessel001"
→ consignment-cli git:(feature/part1) ✕
→ consignment-cli git:(feature/part1) ✕ go run cli.go
2018/05/12 05:31:21 created: true
2018/05/12 05:31:21 description:"This is a test consignment" weight:550 containers:<customer_id:"cust001" origin:"Manchester, United Kingdom" user_id:"user001" > vessel_id:"vessel001"
2018/05/12 05:31:21 description:"This is a test consignment" weight:550 containers:<customer_id:"cust001" origin:"Manchester, United Kingdom" user_id:"user001" > vessel_id:"vessel001"
→ consignment-cli git:(feature/part1) ✕ _
```

至此，我们使用protobuf和grpc创建了一个微服务以及一个客户端。

在下一篇文章中，我们将介绍使用 `go-micro` 框架，以及创建我们的第二个微服务。同时在下一篇文章中，我们将介绍如何容Docker来容器化我们的微服务。

🔖 微服务

