

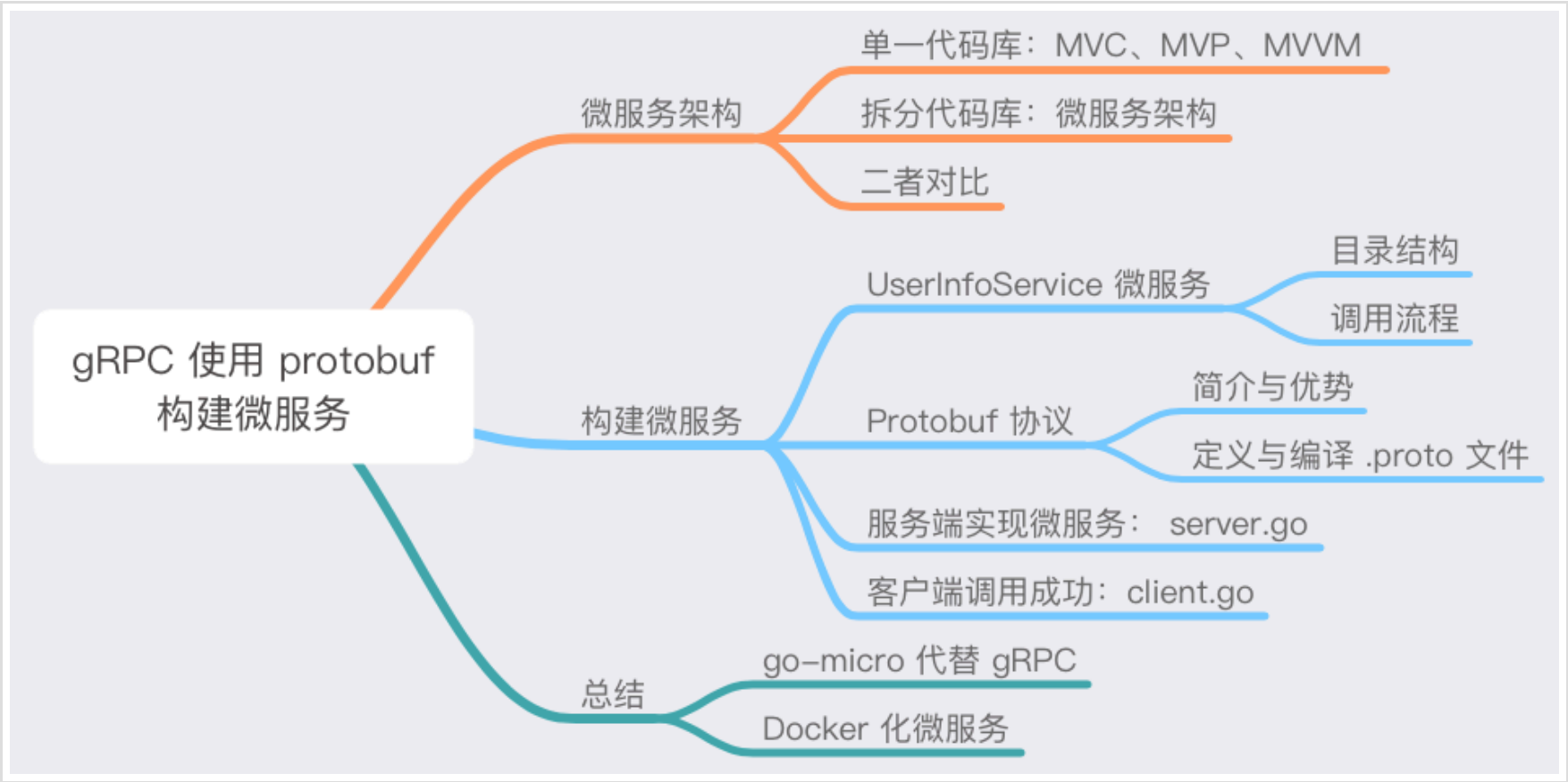
# gRPC 使用 protobuf 构建微服务

📅 2018-05-02 | 👁 1305 | 📄 1,805

gRPC 使用 protobuf 通信构建微服务，本文代码：[GitHub](#)

参考：[用GoLang实现微服务（一）](#)

本文目录：



## 微服务架构

### 单一的代码库

以前使用 Laravel 做 web 项目时，是根据 MVC 去划分目录结构的，即 Controller 层处理业务逻辑，Model 层处理数据库的 CURD，View 层处理数据渲染与页面交互。以及 MVP、MVVM 都是将整个项目的代码是集中在一个代码库中，进行业务处理。这种单一聚合代码的方式在前期实现业务的速度很快，但在后期会暴露很多问题：

- 开发与维护困难：随着业务复杂度的增加，代码的耦合度往往会变高，多个模块相互耦合后不易横向扩展
- 效率和可靠性低：过大的代码量将降低响应速度，应用潜在的安全问题也会累积

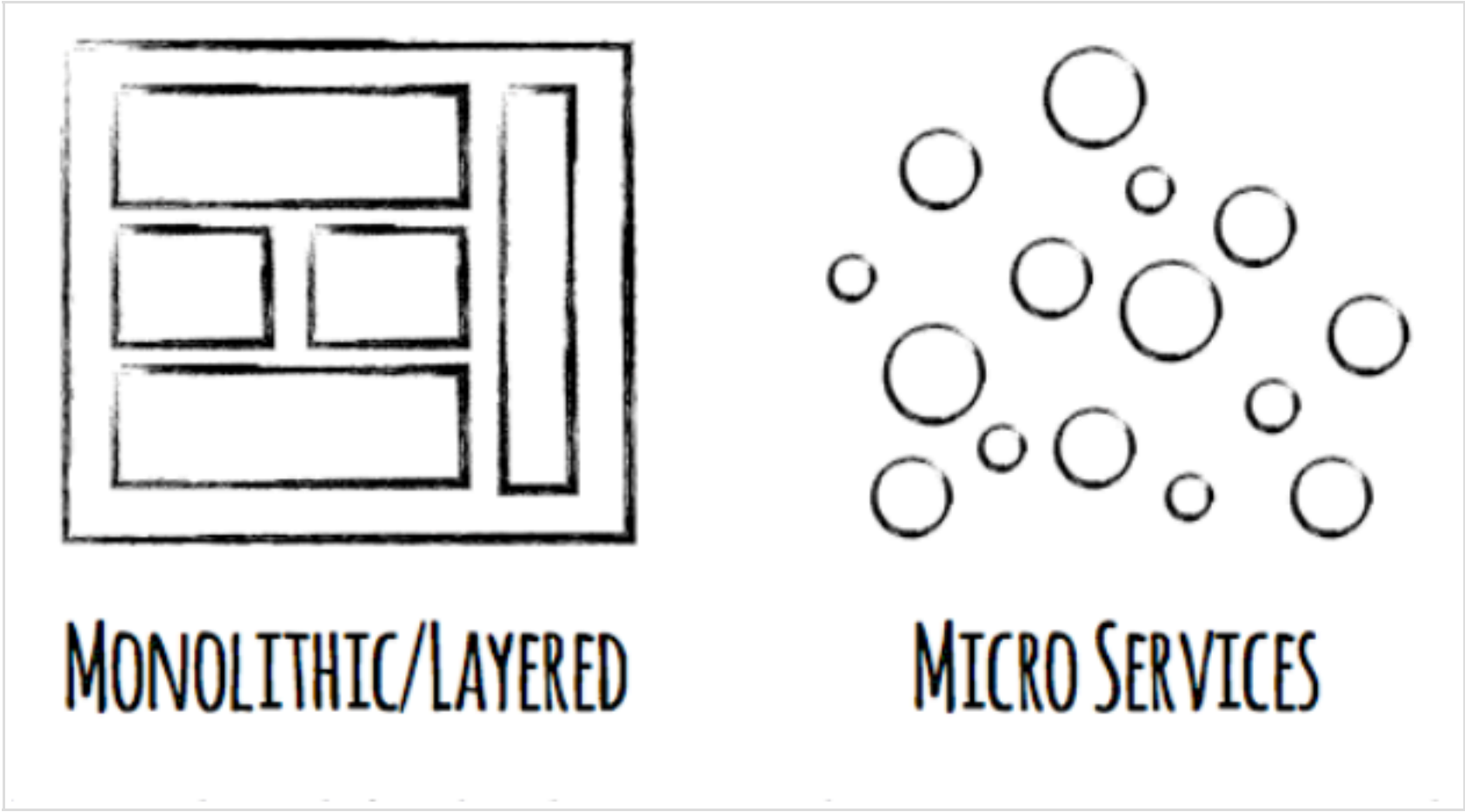
# 拆分的代码库

微服务是一种软件架构，它将一个大且聚合的业务项目拆解为多个小且独立的业务模块，模块即服务，各服务间使用高效的协议（protobuf、JSON 等）相互调用即是 RPC。这种拆分代码库的方式有以下特点：

- 每个服务应作为小规模、独立的业务模块在运行，类似 Unix 的 Do one thing and do it well
- 每个服务应在进行自动化测试和（分布式）部署时，不影响其他服务
- 每个服务内部进行细致的错误检查和处理，提高了健壮性

## 二者对比

本质上，二者只是聚合与拆分代码的方式不同。



参考：[微服务架构的优势与不足](#)

## 构建微服务

### UserInfoService 微服务

接下来创建一个处理用户信息的微服务：UserInfoService，客户端通过 name 向服务端查询用户的年龄、职位等详细信息，需先安装 gRPC 与 protoc 编译器：

```
1 go get -u google.golang.org/grpc
2 go get -u github.com/golang/protobuf/protoc-gen-go
```

## 目录结构

```
1 |— proto
2 |   |— user.proto           // 定义客户端请求、服务端响应的数据格式
3 |   └— user.pb.go          // protoc 为 gRPC 生成的读写数据的函数
4 |— server.go                // 实现微服务的服务端
5 └— client.go                // 调用微服务的客户端
```

## 调用流程



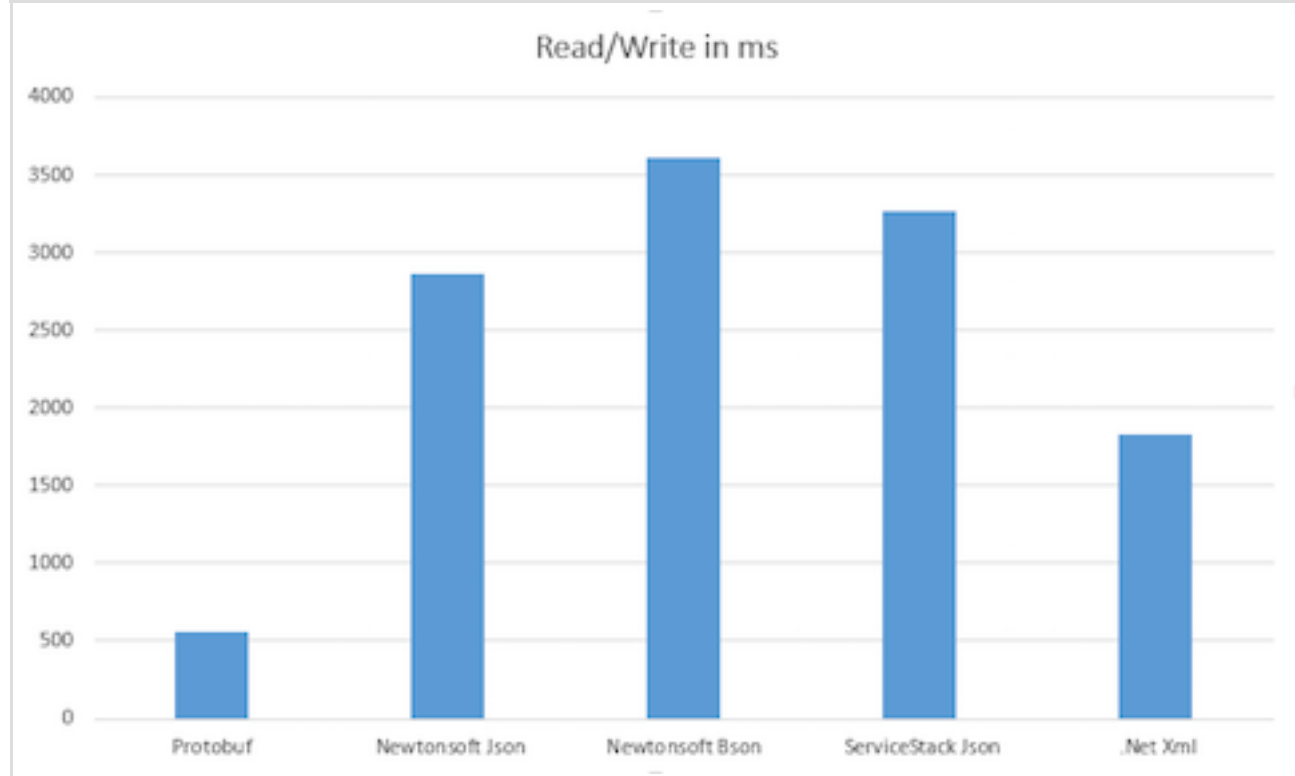
## Protobuf 协议

每个微服务有自己独立的代码库，各自之间在通信时需要高效的协议，要遵循一定的数据结构来解析和编码要传输的数据，在微服务中常使用 `protobuf` 来定义。

Protobuf (protocol buffers) 是谷歌推出的一种二进制数据编码格式，相比 XML 和 JSON 的文本数据编码格式更有优势：

### 读写更快、文件体积更小

它没有 XML 的标签名或 JSON 的字段名，更为轻量，[更多参考](#)



## 语言中立

只需定义一份 .proto 文件，即可使用各语言对应的 protobuf 编译器对其编译，生成的文件中有对 message 编码、解码的函数

### 对于 JSON

- 在 PHP 中需使用 `json_encode()` 和 `json_decode()` 去编解码，在 Golang 中需使用 json 标准库的 `Marshal()` 和 `Unmarshal()` ... 每次解析和编码比较繁琐
- 优点：可读性好、开发成本低
- 缺点：相比 protobuf 的读写速度更慢、存储空间更多

### 对于 Protobuf

- .proto 可生成 .php 或 \*.pb.go ... 在项目中可直接引用该文件中编译器生成的编码、解码函数
- 优点：高效轻量、一处定义多处使用
- 缺点：可读性差、开发成本高

## 定义微服务的 user.proto 文件

```
1 syntax = "proto3";           // 指定语法格式，注意 proto3 不再支持 proto2 的 required 和
2 package proto;               // 指定生成的 user.pb.go 的包名，防止命名冲突
3
4
5 // service 定义开放调用的服务，即 UserInfoService 微服务
6 service UserInfoService {
7     // rpc 定义服务内的 GetUserInfo 远程调用
8     rpc GetUserInfo (UserRequest) returns (UserResponse) {
9     }
```

```

10 }
11
12
13 // message 对应生成代码的 struct
14 // 定义客户端请求的数据格式
15 message UserRequest {
16     // [修饰符] 类型 字段名 = 标识符;
17     string name = 1;
18 }
19
20
21 // 定义服务端响应的数据格式
22 message UserResponse {
23     int32 id = 1;
24     string name = 2;
25     int32 age = 3;
26     repeated string title = 4; // repeated 修饰符表示字段是可变数组, 即 slice 类型
27 }

```

## 编译 user.proto 文件

```

1 # protoc 编译器的 grpc 插件会处理 service 字段定义的 UserInfoService
2 # 使 service 能编码、解码 message
3 $ protoc -I . --go_out=plugins=grpc:. ./user.proto

```

## 生成 user.pb.go

```

1 package proto
2
3 import (
4     context "golang.org/x/net/context"
5     grpc "google.golang.org/grpc"
6 )
7
8 // 请求结构
9 type UserRequest struct {
10     Name string `protobuf:"bytes,1,opt,name=name" json:"name,omitempty"`
11 }
12
13 // 为字段自动生成的 Getter
14 func (m *UserRequest) GetName() string {
15     if m != nil {
16         return m.Name
17     }
18     return ""

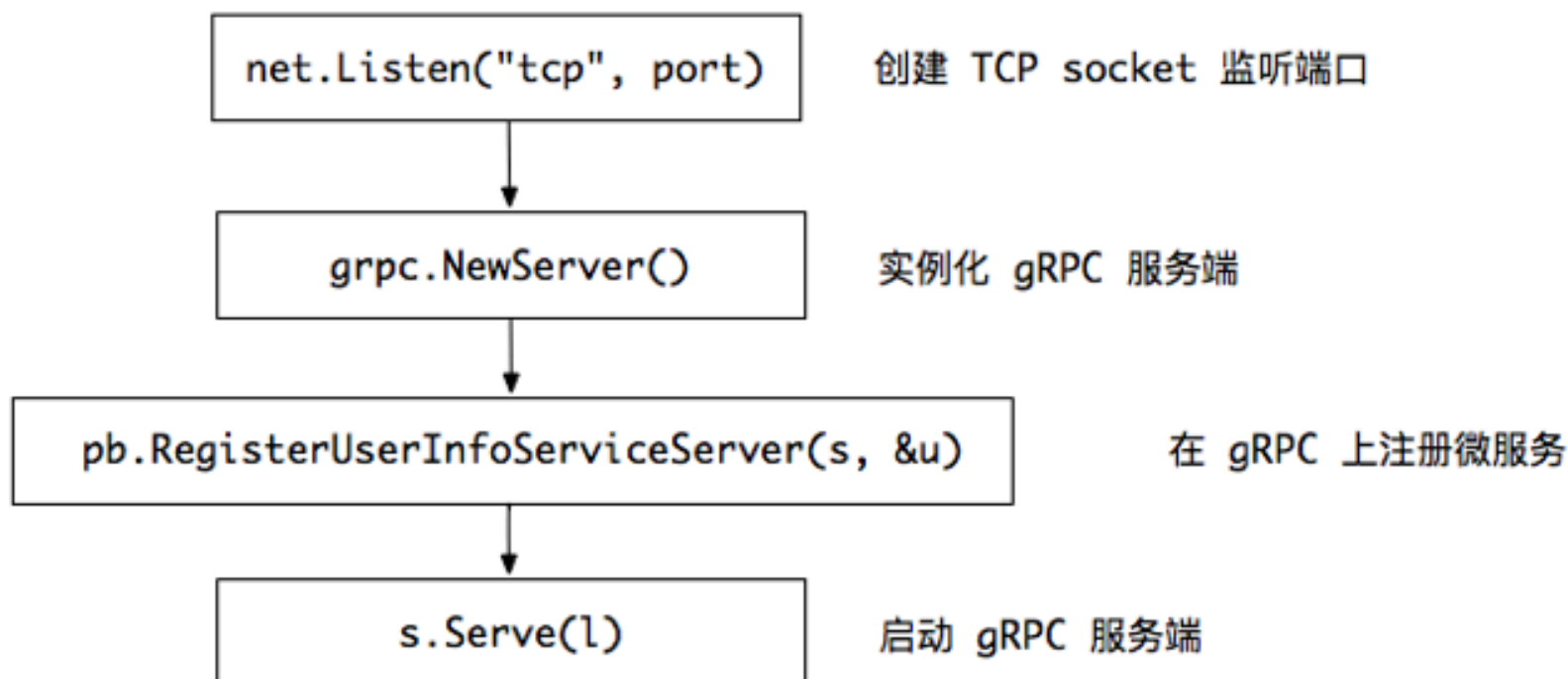
```

```
19 }
20
21 // 响应结构
22 type UserResponse struct {
23     Id    int32    `protobuf:"varint,1,opt,name=id" json:"id,omitempty"`
24     Name  string    `protobuf:"bytes,2,opt,name=name" json:"name,omitempty"`
25     Age   int32    `protobuf:"varint,3,opt,name=age" json:"age,omitempty"`
26     Title []string  `protobuf:"bytes,4,rep,name=title" json:"title,omitempty"`
27 }
28 // ...
29
30 // 客户端需实现的接口
31 type UserInfoServiceClient interface {
32     GetUserInfo(ctx context.Context, in *UserRequest, opts ...grpc.CallOption) (*UserResponse, error)
33 }
34
35
36 // 服务端需实现的接口
37 type UserInfoServiceServer interface {
38     GetUserInfo(context.Context, *UserRequest) (*UserResponse, error)
39 }
40
41 // 将微服务注册到 grpc
42 func RegisterUserInfoServiceServer(s *grpc.Server, srv UserInfoServiceServer) {
43     s.RegisterService(&_UserInfoService_serviceDesc, srv)
44 }
45
46 // 处理请求
47 func _UserInfoService_GetUserInfo_Handler(srv interface{}, ctx context.Context, req interface{}, info *grpc.UnaryInfo, handler func(context.Context, *UserRequest) (*UserResponse, error)) {
48     handler(ctx, req)
49 }
```

## 服务端实现微服务

### 实现流程

server.go 中的 UserInfoService struct  
实现  
user.pb.go 中的 UserInfoServiceServer interface



## 代码参考

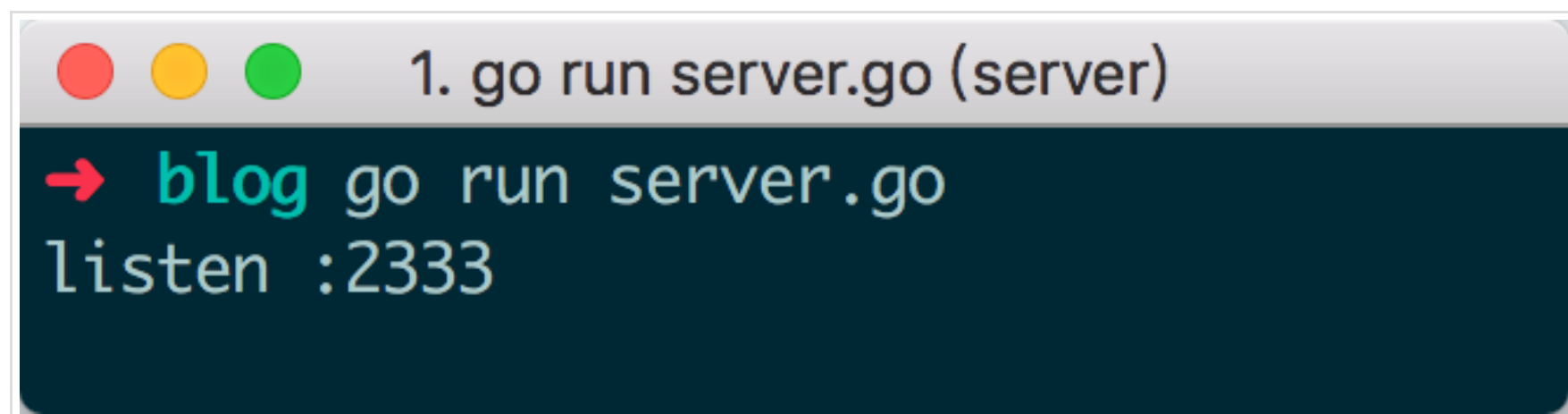
```
1 package main
2 import (...)
3
4 // 定义服务端实现约定的接口
5 type UserInfoService struct{}
6 var u = UserInfoService{}
7
8 // 实现 interface
9 func (s *UserInfoService) GetUserInfo(ctx context.Context, req *pb.UserRequest
10     name := req.Name
11
12     // 模拟在数据库中查找用户信息
13     // ...
14     if name == "wuYin" {
15         resp = &pb.UserResponse{
16             Id:    233,
17             Name:   name,
18             Age:    20,
19             Title: []string{"Gopher", "PHPer"}, // repeated 字段是
20         }
21     }
22     err = nil
23     return
24 }
```

```

25
26 func main() {
27     port := ":2333"
28     l, err := net.Listen("tcp", port)
29     if err != nil {
30         log.Fatalf("listen error: %v\n", err)
31     }
32     fmt.Printf("listen %s\n", port)
33     s := grpc.NewServer()
34
35     // 将 UserInfoService 注册到 gRPC
36     // 注意第二个参数 UserInfoServiceServer 是接口类型的变量
37     // 需要取地址传参
38     pb.RegisterUserInfoServiceServer(s, &u)
39     s.Serve(l)
40 }

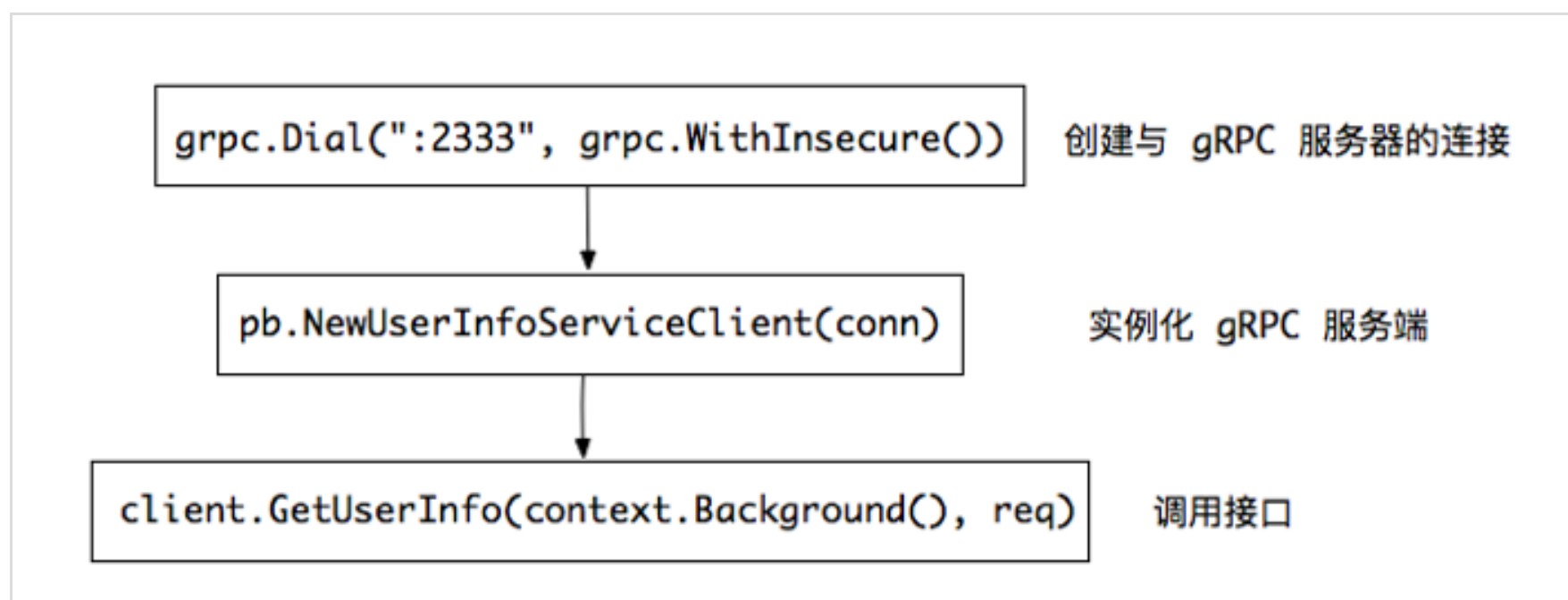
```

运行监听：



客户端调用

实现流程

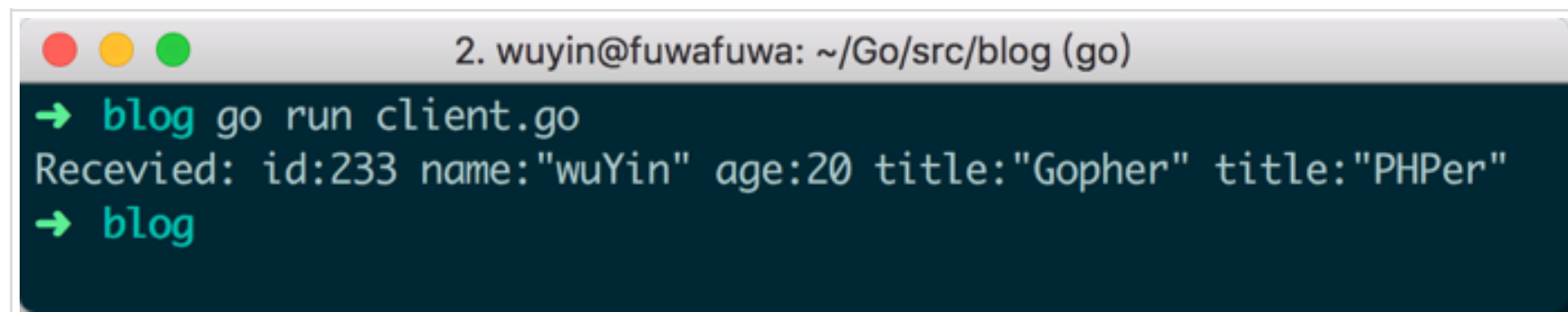




## 代码参考

```
1  package main
2  import (...)
3
4  func main() {
5      conn, err := grpc.Dial(":2333", grpc.WithInsecure())
6      if err != nil {
7          log.Fatalf("dial error: %v\n", err)
8      }
9      defer conn.Close()
10
11     // 实例化 UserInfoService 微服务的客户端
12     client := pb.NewUserInfoServiceClient(conn)
13
14     // 调用服务
15     req := new(pb.UserRequest)
16     req.Name = "wuYin"
17     resp, err := client.GetUserInfo(context.Background(), req)
18     if err != nil {
19         log.Fatalf("resp error: %v\n", err)
20     }
21
22     fmt.Printf("Recevierd: %v\n", resp)
23 }
```

运行调用成功：

A terminal window with a dark background and light green text. The title bar shows the window name '2. wuyin@fuwafuwa: ~/Go/src/blog (go)'. The terminal content shows a prompt 'blog' followed by the command 'go run client.go'. The output is 'Recevierd: id:233 name:"wuYin" age:20 title:"Gopher" title:"PHPer"'. Another prompt 'blog' is visible at the bottom.

```
2. wuyin@fuwafuwa: ~/Go/src/blog (go)
→ blog go run client.go
Recevierd: id:233 name:"wuYin" age:20 title:"Gopher" title:"PHPer"
→ blog
```

## 总结

在上边 UserInfoService 微服务的实现过程中，会发现每个微服务都需要自己管理服务端监听端口，客户端连接后调用，当有很多个微服务时端口的管理会比较麻烦，相比 gRPC，[go-micro](#) 实现了服务发现（Service Discovery）来方便的管理微服务，下节将随服务的 Docker 化一起学习。

更多参考：[Nginx 的微服务系列教程](#)

© 2018  wuYin

访问用户 人 | | 访问量 次