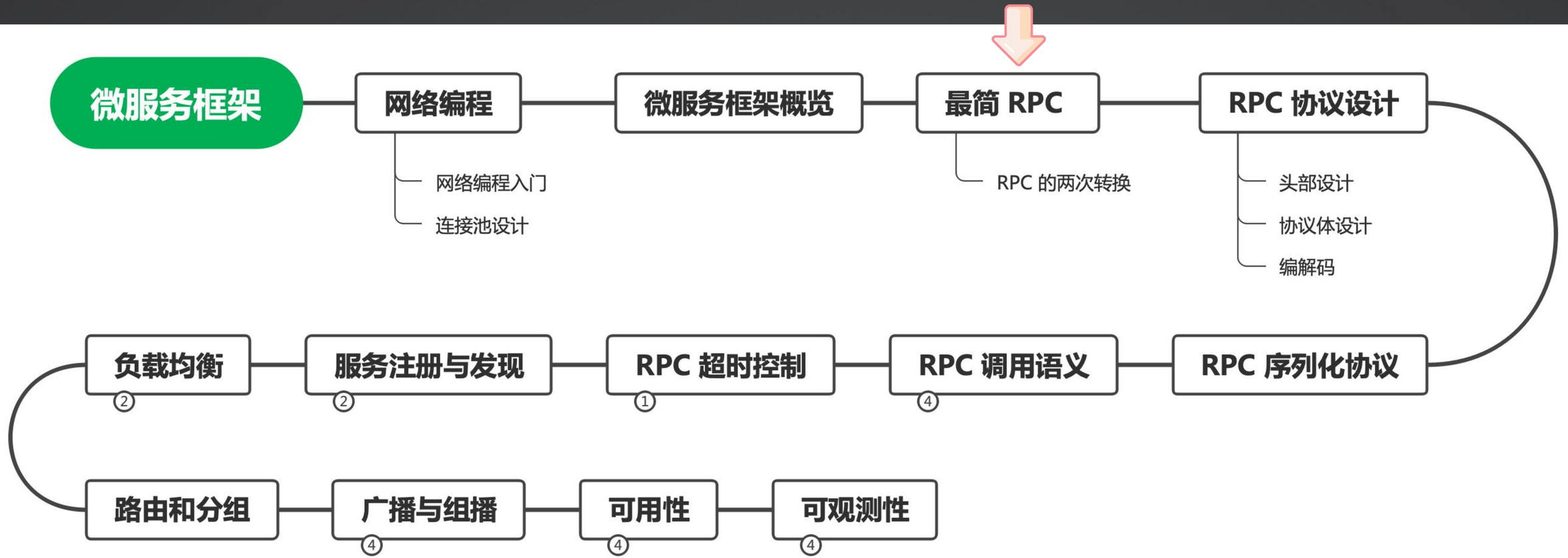
微服务框架—— 最简 RPC

大明

学习路线

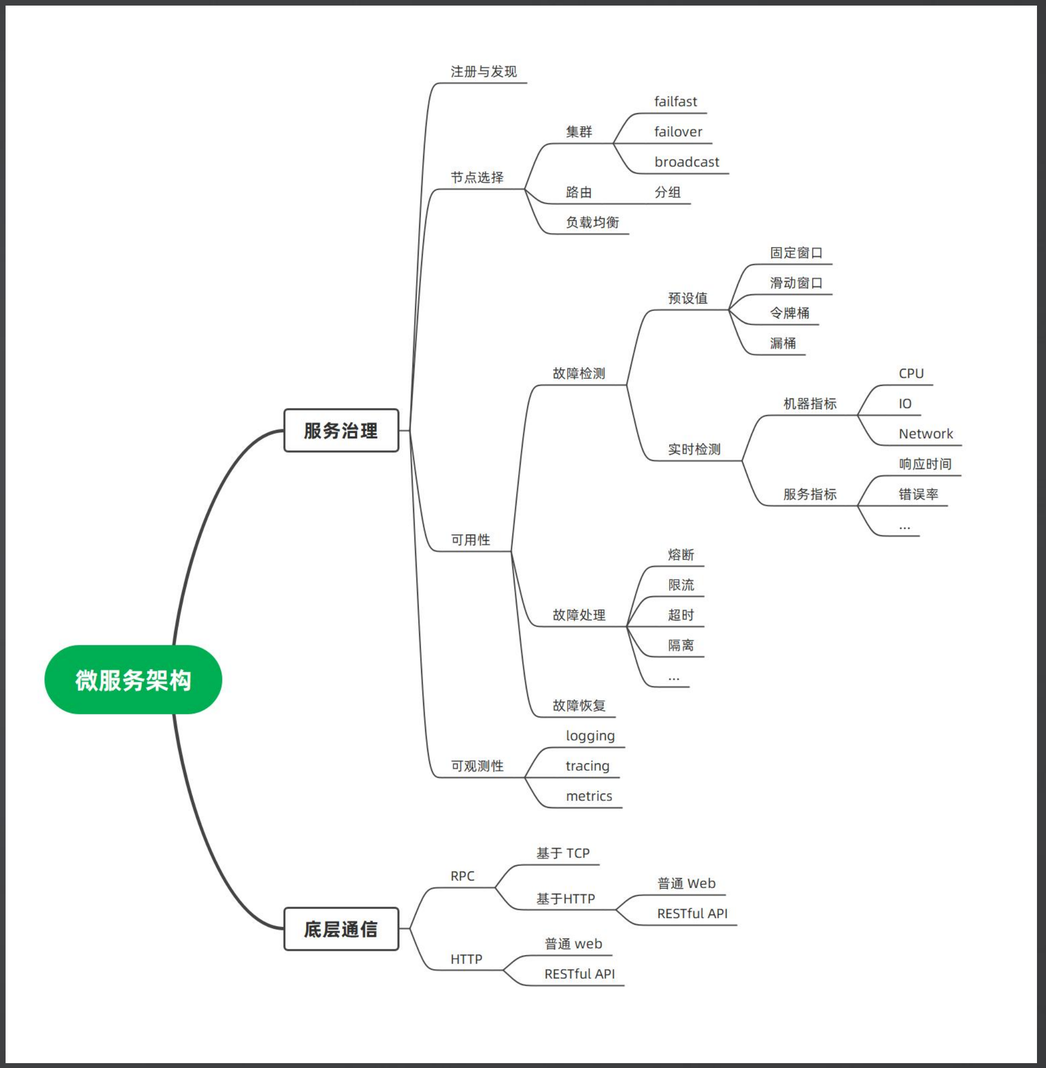

微服务框架
网络编程
网络编程入门
连接池设计
微服务框架概览
最简 RPC
RPC 的两次转换
RPC 协议设计
头部设计
协议体设计
编解码
RPC 序列化协议
RPC 调用语义
4
RPC 超时控制
1
服务注册与发现
2
负载均衡
2
路由和分组
广播与组播
4
可用性
4
可观测性
4

底层通信选型

设计一个微服务框架的起始点，是确定底层通信的方式。因为上层的服务治理，都是建立在这个基础之上。

可行的选择是：

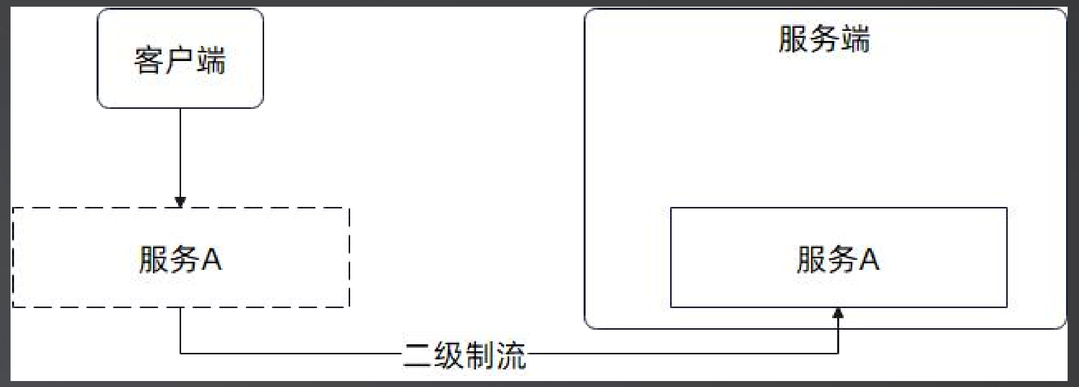
- 直接使用 gRPC
- 直接使用 HTTP
- 设计自己的 RPC 协议：我们的课程会设计自己的协议，但是实践中我建议大家直接使用 gRPC 或者 HTTP



RPC 要解决的核心问题

RPC 的全称是 Remote Procedure Call，即远程过程调用。

核心就是：如同本地调用一般调用服务器上的方法。



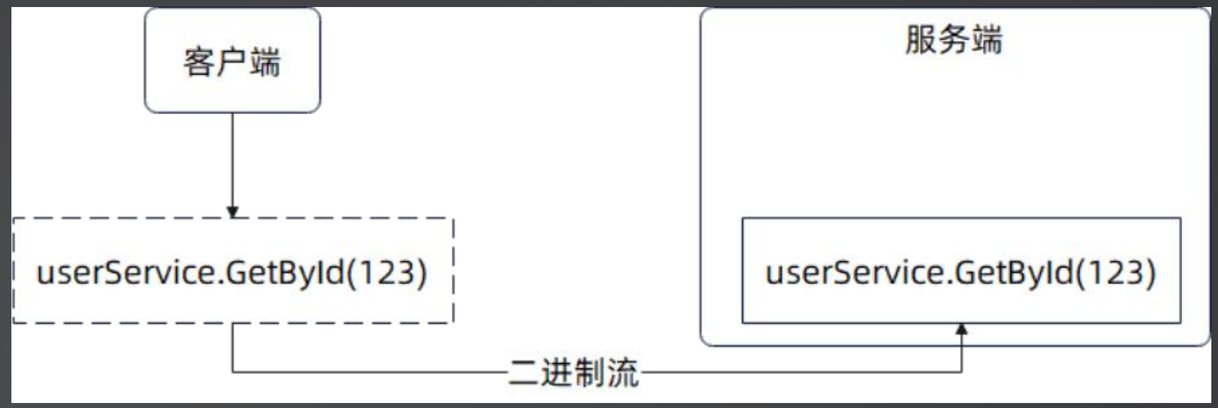
因此要解决的问题就是：怎么把左边虚线框框的服务 A 映射过去右边实线的服务 A。

调用信息

要完成这种映射，首先要解决第一个问题：映射什么？

举个例子，假如我们在客户端调用的是 userService.GetById，传入的参数是 int 类型的值 123。

那么服务端究竟怎么知道客户端调用的是 userService.GetById，参数是 int 类型的 123？



答案很简单：我们把这些信息传过去给服务端，这些信息我们统称为调用信息。

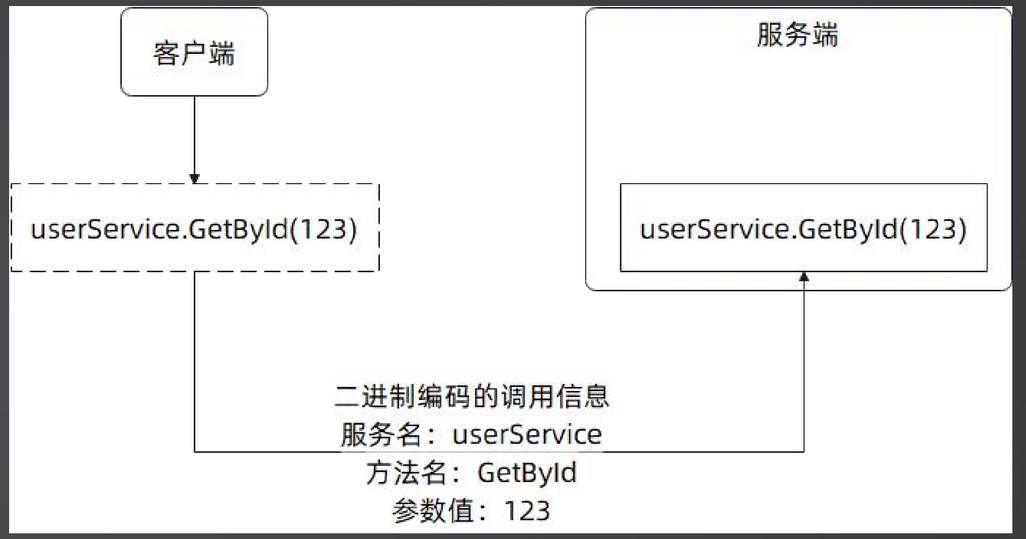
调用信息

那么调用信息需要包含什么？

- 服务名：也就是 userService
- 方法名：GetById
- 参数值：123

要不要参数类型？

如果你在支持重载的语言上设计微服务框架，并且决定支持重载，那么你就需要传递参数类型，否则就不需要。



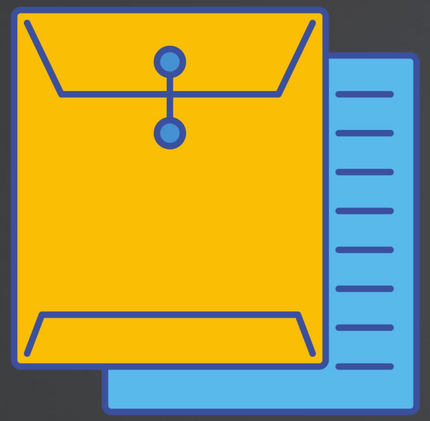
客户端捕捉本地调用

既然要传递调用信息，那么问题就在于：RPC 客户端怎么获得这些调用信息？

这个问题是指：用户调用的是 userService.GetById(123)，我底层框架怎么知道 userService, GetById, 123 这种信息？

假如说是你是 RPC 设计者，你觉得你怎么样才能捕捉到本地调用的信息？

客户端捕捉本地调用



代码生成策略，如 gRPC、go-micro

代理机制，如 Dubbo

代理模式

这里就要用到代理模式了：定义一个结构体，为结构体里面的方法类型字段，注入调用逻辑。

要记住，Go 是没有办法修改方法实现的，所以我们只能迂回救国。

```go
// UserService 声明服务，反射会把 GetById 转成一个 RPC 调用
type UserService struct {
    // 需要注意，因为我们没有办法修改方法的实现，所以我们只能考虑使用这种形态
    GetById func(ctx context.Context, req *FindByUserIdReq) (*FindByUserIdResp, error)
}
```

我们约定：

- 每一个方法第一个参数必须是 context.Context，第二个就是请求结构体指针，并且只有这两个参数。
- 返回值的第一个是响应，并且必须是指针，第二个是 error，并且只有这两个返回值。

这种限制主要就是为了简化微服务框架的代码，在真实生产里面，你可以保持这个限制，也可以考虑去掉。

反射生成代理

```go
func InitClientProxy(address string, val Service) error {
    c, err := NewClient(address)
    if err != nil { return err }
    setFuncField(val, c)
    return nil
}

func setFuncField(val Service, c Proxy) {
    v := reflect.ValueOf(val)
    ele := v.Elem()
    t := ele.Type()

    numField := t.NumField()
    for i := 0; i < numField; i++ {
        field := t.Field(i)
        fieldValue := ele.Field(i)
        if fieldValue.CanSet() {
            fn := func(args []reflect.Value) (results []reflect.Value) {...}
            fieldValue.Set(reflect.MakeFunc(field.Type, fn))
        }
    }
```

这里就是捕捉本地调用
而后调用 Set 方法篡改了它
改成发起 RPC 调用

反射生成代理

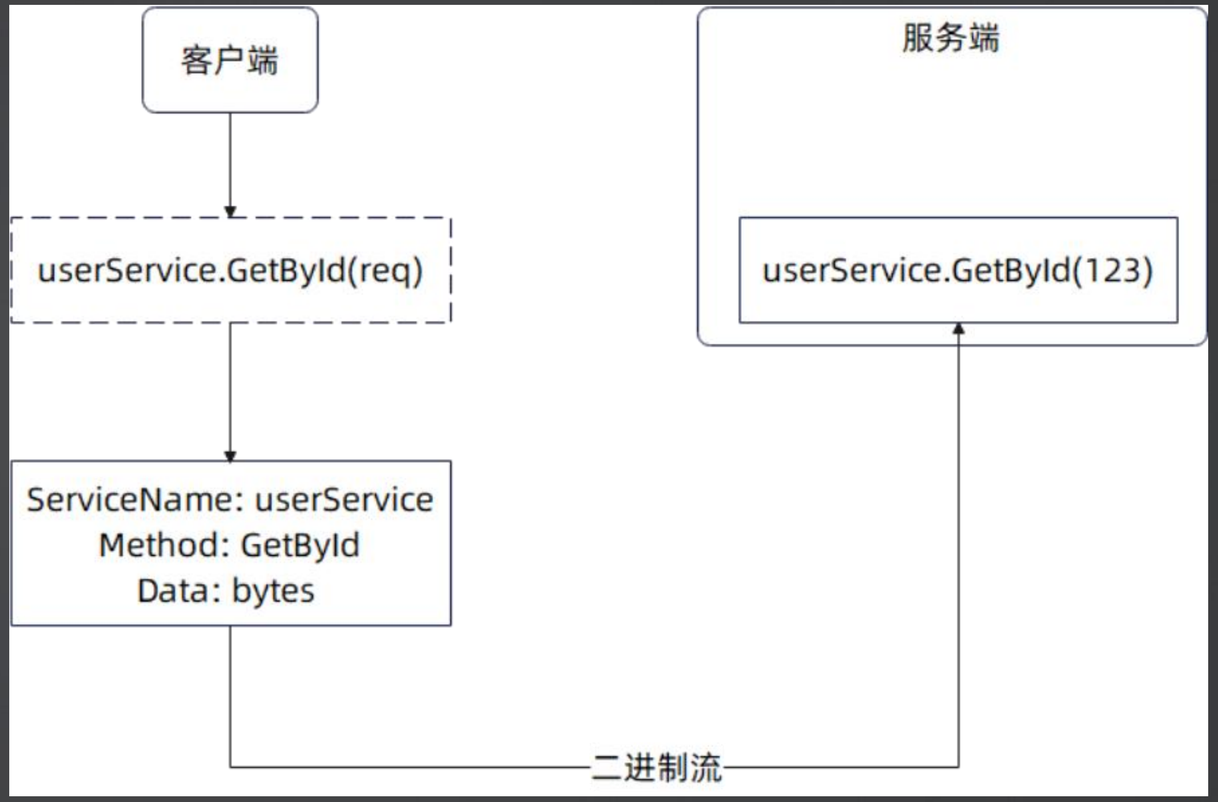
```
fn := func(args []reflect.Value) (results []reflect.Value) {
    in := args[1].Interface()
    out := reflect.New(field.Type.Out( i: 0).Elem()).Interface()
    inData, err := json.Marshal(in)
    if err != nil {
        return []reflect.Value{reflect.ValueOf(out), reflect.ValueOf(err)}
    }

    req := &Request{
        ServiceName: val.ServiceName(),
        Method:      field.Name,
        Data:        inData,
    }

    // 要在下面考虑发过去
```

要在后面考虑怎么把 req 转化为字节流，并且发送过去远端

编码并发送请求

到目前我们已经获得了调用信息，那么就需要将这个调用信息转化为字节，然后通过网络发送到服务端。

实际上这里面就涉及到 RPC 协议设计，以及序列化协议选择的问题。



编码并发送

这里直接写死使用 JSON 来作为序列化协议。

并且我们在请求的基础上，额外增加了一个长度字段，因为服务端需要知道一个消息有多长。

```
    cn := conn.(net.Conn)
    bs, err := json.Marshal(req)          转成 JSON
    if err != nil {
        return nil, fmt.Errorf("client: 无法序列化请求, #{err}")
    }

    encode := EncodeMsg(bs)               加上长度字段
    _, err = cn.(net.Conn).Write(encode)
    if err != nil : nil, err

    bs, err = ReadMsg(cn.(net.Conn))      读取响应
    if err != nil {
        return nil, fmt.Errorf("client: 无法读取响应 #{err}")
    }

    resp := &Response{}
    err = json.Unmarshal(bs, resp)
    return resp, nil
}
```

服务端接收

```go
func (s *Server) handleConnection(conn net.Conn) {
    for {
        bytes, err := ReadMsg(conn)
        if err != nil { ... }
        // go func() {
        u := &Request{}
        err = json.Unmarshal(bytes, u)
        resp, er := s.Invoke(context.Background(), u)
        if resp == nil {
            resp = &Response{}
        }
        if er != nil && len(resp.Error) == 0 {
            resp.Error = er.Error()
        }
```

将调用信息还原回来

```go
func ReadMsg(conn net.Conn) (bs []byte, err error) {
    msgLenBytes := make([]byte, lenBytes)
    length, err := conn.Read(msgLenBytes)
    defer func() {
        if msg := recover(); msg != nil {
            err = errors.New(fmt.Sprintf("#{msg}"))
        }
    }()
    if err != nil { return nil, err }

    if length != lenBytes { return nil, errors.New("could not

    dataLen := binary.BigEndian.Uint64(msgLenBytes)
    bs = make([]byte, dataLen)
    _, err = io.ReadFull(conn, bs)
    return bs, err
}
```

服务端执行逻辑

```go
func (s *Server) Invoke(ctx context.Context, req *Request) (*Re
    resp := &Response{}
    service, ok := s.services[req.ServiceName]
    if !ok {
        return resp, fmt.Errorf( format: "server: 未找到服务, 服务
    }
    respData, err := service.invoke(ctx, req.Method, req.Data)
    if err != nil {
        return resp, err
    }
    resp.Data = respData
    return resp, nil
}
```

根据服务名查找服务

```go
func (s *reflectionStub) invoke(ctx context.Context, methodName st
    method := s.value.MethodByName(methodName)
    inType := method.Type().In( i: 1)
    in := reflect.New(inType.Elem())
    err := json.Unmarshal(data, in.Interface())
    if err != nil {
        return nil, err
    }
    res := method.Call([]reflect.Value{reflect.ValueOf(ctx), in})
    if len(res) > 1 && !res[1].IsZero() {
        return nil, res[1].Interface().(error)
    }
    return json.Marshal(res[0].Interface())
}
```

反射执行方法

最简 RPC 总结

客户端：

- 初始化代理
- 代理会利用反射获得调用信息
- 将调用信息编码成字节流，加上长度字段
- 将数据发送到服务端
- 等待并且解析响应

服务端：

- 启动服务器监听端口
- 接收连接，并且读取数据
- 将数据还原回调用信息
- 根据服务名查找该实例上注册的服务
- 利用反射执行方法调用
- 写回响应

面试要点

- 什么是 RPC？远程过程调用，类似的还有 RMI：远程方法调用。
- 和使用 HTTP 接口比起来，使用 RPC 有什么优势？不必关心 HTTP 调用的细节，对于使用者来说就如同本地调用一般。
- RPC 框架的要点是什么？客户端捕捉调用信息，编码成二进制，发送到服务端。服务端查找本地服务，执行调用，写回响应。任何一个 RPC 框架都类似。所以如果面试官问 gRPC 的大概步骤，也可以这么回答。
- RPC 框架怎么捕捉本地调用信息（或者说RPC框架怎么知道你调用的是什么）？主要依赖于代理模式和代码生成技术。
- 什么是代理模式？什么是动态代理模式？动态代理可以看做是动态生成的代理，一般是指运行时生成的代理。
- 动态代理技术能用来做什么？四个字，为所欲为。在这里就是用来发起 RPC 调用，然后再返回响应。

Q & A

THANKS