

## 41 | 实战（一）：“画图”程序后端实战

2019-09-13 许式伟

许式伟的架构课

[进入课程 >](#)



讲述：姚迪迈

时长 10:22 大小 9.50M



你好，我是七牛云许式伟。

到今天为止，服务端开发的基本内容已经讲完了。我们花了比较长的篇幅来介绍服务端的基础软件，包括负载均衡和各类存储中间件。然后我们上一讲介绍了服务端在业务架构上的一些通用问题。

今天我们开始进入实战。

对比服务端和桌面的内容可以看出，服务端开发和桌面端开发各自有各自的复杂性。服务端开发，难在基础软件很多，对程序员和架构师的知识面和理解深度都有较高的要求。但从业务复杂性来说，服务端的业务逻辑相对简单。而桌面端开发则相反，它的难点在于用户交互逻辑复杂，代码量大，业务架构的复杂性高。

上一章的实战篇，蛮多人反馈有点难，这某种程度来说和我们课程内容设计的规划有关。上一章我们从架构角度来说，偏重于介绍概要设计，也就是系统架构。所以我们对实现细节并没有做过多的剖析，而是把重心放在模块之间的接口耦合上。这是希望你把关注点放在全局，而不是一上来就进入局部细节。但是由于缺乏完整流程的剖析，大家没法把整个过程串起来，理解上就会打折扣。

这一章我们在架构上会偏重于详细设计。这在实战篇也会有所体现。

在上一章，我们实现了一个 mock 版本的服务端，代码如下：

<https://github.com/qiniu/qpaint/tree/v31/paintdom>

接下来我们一步步把它变成一个产品级的服务端程序。

## RPC 框架

第一步，我们引入 RPC 框架。

为了方便你理解，在上一章的实战中，我们的 mock 服务端程序没有引入任何非标准库的内容。代码如下：

<https://github.com/qiniu/qpaint/blob/v31/paintdom/service.go>


整个 Service 大约 280 行代码。

我们改为基于七牛云开源的 [restrpc](#) 框架来实现，代码如下：

<https://github.com/qiniu/qpaint/blob/v41/paintdom/service.go>

这样，整个 Service 就大约只剩下 163 行代码，只有原先的 60% 不到。

到底少写了哪些代码？我们拿创建一个新图形来看下。原先我们这样写：

 复制代码


```
1 func (p *Service) PostShapes(w http.ResponseWriter, req *http.Request, args []string) {  
2     id := args[0]
```

```

3      drawing, err := p.doc.Get(id)
4      if err != nil {
5          ReplyError(w, err)
6          return
7      }
8
9      var aShape serviceShape
10     err = json.NewDecoder(req.Body).Decode(&aShape)
11     if err != nil {
12         ReplyError(w, err)
13         return
14     }
15
16     err = drawing.Add(aShape.Get())
17     if err != nil {
18         ReplyError(w, err)
19         return
20     }
21     ReplyCode(w, 200)
22 }

```

现在这样写：

 复制代码


```

1 func (p *Service) PostShapes(aShape *serviceShape, env *restrpc.Env) (err error) {
2     id := env.Args[0]
3     drawing, err := p.doc.Get(id)
4     if err != nil {
5         return
6     }
7     return drawing.Add(aShape.Get())
8 }

```

这个例子返回包比较简单，没有 HTTP 包的正文。

我们再来看一个返回包比较复杂的例子，取图形的内容。原先我们这样写：

 复制代码

```

1 func (p *Service) GetShape(w http.ResponseWriter, req *http.Request, args []string) {
2     id := args[0]
3     drawing, err := p.doc.Get(id)
4     if err != nil {


```

```

5         ReplyError(w, err)
6         return
7     }
8
9     shapeID := args[1]
10    shape, err := drawing.Get(shapeID)
11    if err != nil {
12        ReplyError(w, err)
13        return
14    }
15    Reply(w, 200, shape)
16 }

```

现在这样写：

 复制代码

```

1 func (p *Service) GetShape(env *restrpc.Env) (shape Shape, err error) {
2     id := env.Args[0]
3     drawing, err := p.doc.Get(id)
4     if err != nil {
5         return
6     }
7
8     shapeID := env.Args[1]
9     return drawing.Get(shapeID)
10 }

```

对比这两个例子，我们可以看出：

原先这两个请求 `POST /drawings/<DrawingID>/shapes`、`GET /drawings/<DrawingID>/shapes/<ShapeID>` 中的 URL 参数如 `DrawingID`、`ShapeID` 的值，是通过参数 `args[0]`、`args[1]` 传入，现在通过 `env.Args[0]`、`env.Args[1]` 传入。


原先我们 `PostShapes` 需要自己定义 `Shape` 实例并解析 HTTP 请求包 `req.Body` 的内容。现在我们只需要在参数中指定 `Shape` 类型，`restrpc` 框架就自动完成参数的解析。

原先我们 `GetShape` 需要自己回复错误或者返回正常的 HTTP 协议包。现在我们只需要在返回值列表中返回要回复的数据，`restrpc` 框架自动完成返回值的序列化并回复 HTTP 请求。

通过对比两个版本的代码差异，我们大体能够猜得出来，restrpc 的 HTTP 处理函数背后都干了些啥。其核心代码如下：

[https://github.com/qiniu/http/blob/v2.0.2/rpcutil/rpc\\_util.go#L96](https://github.com/qiniu/http/blob/v2.0.2/rpcutil/rpc_util.go#L96)

值得关注的是 Env 的支持，RPC 框架并没有限定 Env 类具体是什么样子的，只是规定它需要满足以下接口：

 复制代码

```
1 type itfEnv interface {  
2     OpenEnv(rcvr interface{}, w *http.ResponseWriter, req *http.Request) error  
3     CloseEnv()  
4 }
```

在 OpenEnv 方法中，我们一般进行 Env 的初始化工作。CloseEnv 方法则反之。为什么 OpenEnv 方法中，ResponseWriter 接口是以指针方式传入？因为可能会有客户希望改写 ResponseWriter 的实现。

比如，假设我们要给 RPC 框架扩展 API 审计日志的功能。那么我们就需要接管并记录用户返回的 HTTP 包，这时我们就需要改写 ResponseWriter 以达到接管并记录的目的。

另外值得注意的是，restrpc 版本的 HTTP 请求的处理函数，看起来不再那么像 HTTP 处理函数，倒像一个普通函数。

这意味着我们可以有两种方式来测试 Service 类。除了用正常测试 HTTP Service 的方法来测试它以外，我们也可以把 Service 类当成普通类来测试，这大大降低单元测试的成本。因为我们不用再需要包装服务的 Client SDK，然后再基于 Client SDK 做单元测试。

当然，我们有这样的一种低成本测试方式，但还是会担心这种测试方法可能不能覆盖一些编码上的小意外，毕竟我们没有走 HTTP 协议，心里多多少少有些不踏实。


理解了 restrpc 的 HTTP 处理函数，剩下的就是 restrpc 的路由功能。它是由 restrpc.Router 类的 Register 函数完成的。代码如下：

<https://github.com/qiniu/http/blob/v2.0.1/restrpc/restroute.go#L39>

它支持两种路由方式，一种是根据方法名字自动路由。比如 `POST /drawings/<DrawingID>/shapes` 这样的请求，要求方法名为 `"PostDrawings_Shapes"`。 `GET /drawings/<DrawingID>/shapes/<ShapeID>` 这样的请求，要求方法名为 `"GetDrawings_Shapes_"`。

规则倒是比较简单，路径中的 `"/"` 由单词首字母大写来分隔，URL 参数如 `DrawingID`、`ShapeID` 这些则替换为 `"_"`。

当然有的人会认为这种方法名字看起来很丑。那么就可以选择手工路由的方式，传入 `routeTable`。它看起来是这样的：

 复制代码

```
1 var routeTable = [][]string{
2     {"POST /drawings", "PostDrawings"},
3     {"GET /drawings/*", "GetDrawing"},
4     {"DELETE /drawings/*", "DeleteDrawing"},
5     {"POST /drawings/*/sync", "PostDrawingSync"},
6     {"POST /drawings/*/shapes", "PostShapes"},
7     {"GET /drawings/*/shapes/*", "GetShape"},
8     {"POST /drawings/*/shapes/*", "PostShape"},
9     {"DELETE /drawings/*/shapes/*", "DeleteShape"},
10 }
```

虽然是手工路由，但是方法名仍然有限制，要求必须是 `Get`、`Put`、`Post`、`Delete` 开头。

## 业务逻辑的分层


理解了 `restrpc` 框架，我们再看下 `QPaint` 服务端的业务本身。可以看出，我们的服务端业务逻辑被分为两层：一层是业务逻辑的实现层，通常我们有意识地把它组织为一颗 `DOM` 树。代码如下：

<https://github.com/qiniu/qpaint/blob/v41/paintdom/drawing.go>

<https://github.com/qiniu/qpaint/blob/v41/paintdom/shape.go>

另一层则是 `RESTful API` 层，它负责接收用户的网络请求，并转为对底层 `DOM` 树的方法调用。有了上面我们介绍的 `restrpc` 框架，这一层的每个方法往往都比较简单，甚至有的只

是很简单的一句函数调用。比如：

 复制代码

```
1 func (p *Service) DeleteDrawing(env *restrpc.Env) (err error) {  
2     id := env.Args[0]  
3     return p.doc.Delete(id)  
4 }
```

完整的 RESTful API 层代码如下：

<https://github.com/qiniu/qpaint/blob/v41/paintdom/service.go>

这样分层的原因，是因为我们实现核心业务逻辑的时候，并不会假设一定通过 RESTful API 暴露。我们考虑这样几种可能性：

其一，有可能我们根本不需要网络调用。

做个类比，我们都知道 mysql 是通过 TCP 协议提供服务接口的，而 sqlite 是嵌入式数据库，是通过本地的函数调用提供服务接口的。这里分层就类似于我实现 mysql 的时候，先在底层实现了一个类似 sqlite 的嵌入式数据库，然后再提供基于 TCP 协议的网络接口。

其二，有可能我们需要支持很多种网络协议。

我们今天流行 RESTful API，所以我们的接口是 RESTful 风格的。如果有一天我们像 Github 一样想改用 GraphQL，那么至少底层的业务逻辑实现层是不需要改变的，我们只需要实现相对薄的 GraphQL 层就行了。

而且，往往在这种情况下 RESTful API 和 GraphQL 是需要同时支持的。毕竟我们不可能为了赶时髦，就把老用户弃之不顾了。

在需要同时支持多套网络接口的时候，这种分层的价值就体现出来了，不同网络接口的模块之间，共享了同一份 DOM 树的实例，整个体系不仅实现了多协议并存，还实现了完美的解耦，彼此之间完全独立。

## 单元测试




聊完了业务，我们再来看看单元测试。

之前，我们单元测试基本上没怎么做：

[https://github.com/qiniu/qpaint/blob/v31/paintdom/service\\_test.go#L62](https://github.com/qiniu/qpaint/blob/v31/paintdom/service_test.go#L62)

代码如下：

 复制代码

```
1 type idRet struct {
2     ID string `json:"id"`
3 }
4
5 func TestNewDrawing(t *testing.T) {
6     ...
7     var ret idRet
8     err := Post(&ret, ts.URL + "/drawings", "")
9     if err != nil {
10         t.Fatal("Post /drawings failed:", err)
11     }
12     if ret.ID != "10001" {
13         t.Log("new drawing id:", ret.ID)
14     }
15 }
```

从这里的测试代码可以看出，我们就只是创建了一个 drawing，并且要求返回的 drawingID 为 “10001”。

从单元测试的角度，这样的测试力度当然是非常不足的。同样的测试案例，用我们上一讲介绍的 [httpptest](#) 测试框架实现如下：


 复制代码

```
1 func TestNewDrawing(t *testing.T) {
2     ...
3     ctx := httpptest.New(t)
4     ctx.Exec(
5         `
6         post http://qpaint.com/drawings
7         ret 200
8         json '{"id": "10001"}'
9     `)
```



```
10 }
```

当然，实际我们应该去测试更多的情况，比如：

 复制代码

```
1 func TestService(t *testing.T) {
2     ...
3     ctx := httptest.New(t)
4     ctx.Exec(
5         `
6         post http://qpaint.com/drawings
7         ret 200
8         json '{
9             "id": $(id1)
10        }'
11        match $(line1) '{
12            "id": "1",
13            "line": {
14                "pt1": {"x": 2.0, "y": 3.0},
15                "pt2": {"x": 15.0, "y": 30.0},
16                "style": {
17                    "lineWidth": 3,
18                    "lineColor": "red"
19                }
20            }
21        }'
22        post http://qpaint.com/drawings/$(id1)/shapes
23        json $(line1)
24        ret 200
25        get http://qpaint.com/drawings/$(id1)/shapes/1
26        ret 200
27        json $(line1)
28        `)
29        if !ctx.GetVar("id1").Equal("10001") {
30            t.Fatal(`$(id1) != "10001"`)
31        }
32    }
```

这个案例我们想演示什么？这是一个相对复杂的案例。首先我们创建了一个 drawing，并且将 drawingID 放到变量 `$(id1)` 中。随后，我们向该 drawing 中添加了一条直线 `$(line1)`。为了确认添加成功，我们取出了该图形对象，并且判断取得的图形和添加进去的 `$(line1)` 是否一致。

另外，它也演示了 qiniutest DSL 脚本和 Go 语言代码的互操作性。我们用 Go 代码取得变量 `$ (id1)`，并且判断它是否和 “10001” 相等。

关于 qiniutest 更多的内容，请查阅以下资料：

<https://github.com/qiniu/httpptest>

<https://github.com/qiniu/qiniutest>

演讲稿：<http://open.qiniudn.com/qiniutest.pdf>

在我们的测试代码中，还使用了一个七牛云开源的 mockhttp 组件，它也非常有意思：

<https://github.com/qiniu/x/blob/v8.0.1/mockhttp/mockhttp.go>

这个 mockhttp 并不真去监听端口，感兴趣的同学可以研究一下。

## 结语

我们总结一下今天的内容。

从今天开始我们会一步步将之前写的 mock 服务端改造为真实的服务端程序。

我们第一步改造的是 RPC 框架和单元测试。这样我们第一次开始依赖第三方的代码库，如下：

<http://github.com/qiniu/http>（用到 restrpc）

<http://github.com/qiniu/qiniutest>

<http://github.com/qiniu/x>（用到 mockhttp）

一旦有了外部依赖，我们就需要考虑依赖库的版本管理。好的一点是大多数现代语言都有很好的版本管理规范，对于 Go 语言我们用 go mod 来做版本管理。

如果你对今天的内容有什么思考与解读，欢迎给我留言，我们一起讨论。下一讲开始我们继续实战。

如果你觉得有所收获，也欢迎把文章分享给你的朋友。感谢你的收听，我们下期再见。



# 许式伟的架构课

从源头出发，带你重新理解架构设计

许式伟  
七牛云 CEO



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 40 | 服务端的业务架构建议

## 精选留言 (3)

写留言



**Aaron Cheung**

2019-09-13

今天中秋节 没人打卡了吗

展开 ∨



5



**Charles**

2019-09-16

基础不大好，请教下许老师，这篇中提到的rpc框架作用是为了更快速的开发RESTful API，而封装了底层http协议处理吗？

展开 ∨

作者回复: 是



我来也

2019-09-13

终于又看到了熟悉的golang代码😊

展开

