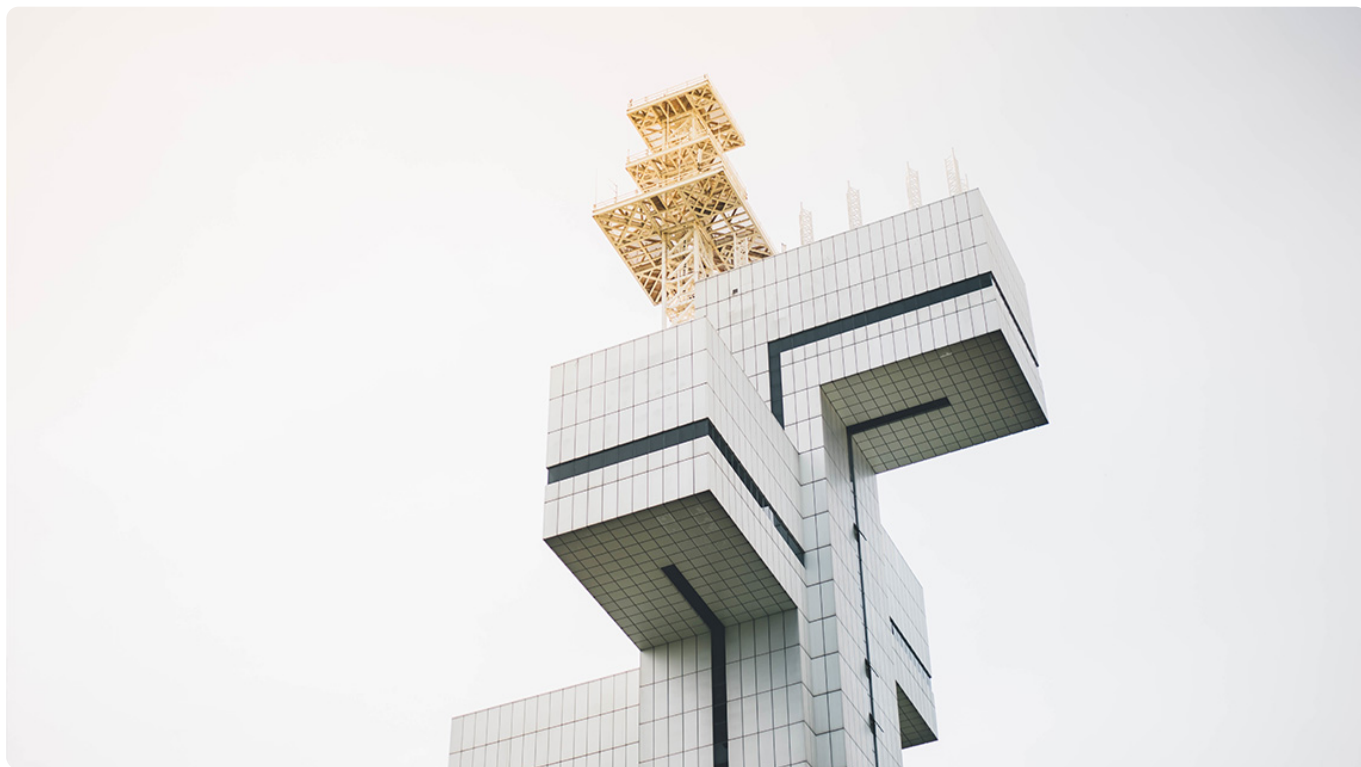


42 | 实战（二）：“画图”程序后端实战

2019-09-17 许式伟

许式伟的架构课

[进入课程 >](#)



讲述：姚迪迈

时长 07:33 大小 6.94M



你好，我是七牛云许式伟。

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>

接下来我们将一步步迭代，把它变成一个产品级的服务端程序。


我们之前已经提到，服务端程序的业务逻辑被分为两层：底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。上层则是 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。

上一讲我们关注的是 RESTful API 层。我们为了实现它，引入了 RPC 框架 [restrpc](#) 和单元测试框架 [qiniutest](#)。

这一讲我们关注的是底层的业务逻辑实现层。

使用界面（接口）

我们先看下这一层的使用界面（接口）。从 DOM 树的角度来说，在这一讲之前，它的逻辑结构如下：

 复制代码

```
1 <Drawing1>
2   <Shape11>
3   ...
4   <Shape1M>
5   ...
6 <DrawingN>
```


从大的层次结构来说只有三层：

Document => Drawing => Shape

那么，在引入多租户（即多用户，每个用户有自己的 uid）之后的 DOM 树，会发生什么样的变化？


比如我们是否应该把它变成四层：

Document => User => Drawing => Shape

 复制代码

```
1 <User1>
2   <Drawing11>
3     <Shape111>
4     ...
5     <Shape11M>
6     ...
7   <Drawing1N>
8   ...
```

我的答案是：多租户不应该影响 DOM 树的结构。所以正确的设计应该是：

 复制代码

```
1 <Drawing1>, 隶属于某个 <uid>
2   <Shape11>
3   ...
4   <Shape1M>
5   ...
6 <DrawingN>, 隶属于某个 <uid>
```


也就是说，多租户只会导致 DOM 树多了一些额外的约定，通常我们应该把它看作某种程度的安全约定，避免访问到没有权限访问到的资源。

所以多租户不会导致 DOM 层级变化，但是它会导致接口方法的变化。比如我们看 Document 类的方法。之前，Document 类接口看起来是这样的：

 复制代码

```
1 func (p *Document) Add() (drawing *Drawing, err error)
2 func (p *Document) Get(dgid string) (drawing *Drawing, err error)
3 func (p *Document) Delete(dgid string) (err error)
```


现在它变成了：

 复制代码

```
1 // Add 创建新 drawing。
2 func (p *Document) Add(uid UserID) (drawing *Drawing, err error)
3
4 // Get 获取 drawing。
5 // 我们会检查要获取的 drawing 是否为该 uid 所拥有，如果不属于则获取会失败。
6 func (p *Document) Get(uid UserID, dgid string) (drawing *Drawing, err error)
7
8 // Delete 删除 drawing。
9 // 我们会检查要删除的 drawing 是否为该 uid 所拥有，如果不属于删除会失败。
10 func (p *Document) Delete(uid UserID, dgid string) (err error)
```

正如注释中说的那样，传入 uid 是一种约束，我们无论是获取还是删除 drawing，都会看这个 drawing 是不是隶属于该用户。


对于 QPaint 程序来说，Document 类之外其他类的接口倒是没有发生变化。比如 Drawing 类的接口如下：

 复制代码

```
1 func (p *Drawing) GetID() string
2 func (p *Drawing) Add(shape Shape) (err error)
3 func (p *Drawing) List() (shapes []Shape, err error)
4 func (p *Drawing) Get(id ShapeID) (shape Shape, err error)
5 func (p *Drawing) Set(id ShapeID, shape Shape) (err error)
6 func (p *Drawing) SetZorder(id ShapeID, zorder string) (err error)
7 func (p *Drawing) Delete(id ShapeID) (err error)
8 func (p *Drawing) Sync(shapes []ShapeID, changes []Shape) (err error)
```

但是这只是因为 QPaint 程序的业务逻辑比较简单。虽然我们需要极力避免接口因为多租户而产生变化，但是这种影响有时候却是不可避免的。

另外，在描述类的使用界面时，我们不能只描述语言层面的约定。比如上面的 Drawing 类，我们引用图形（Shape）对象时，用的是 Go 语言的 interface。如下：

 复制代码

```
1 type ShapeID = string
2
3 type Shape interface {
4     GetID() ShapeID
5 }
```

但是，是不是这一接口就是图形（Shape）的全部约束？

答案显然不是。

我们先看一个最基本的约束：考虑到 Drawing 类的 List 和 Get 返回的 Shape 实例，会被直接作为 RESTful API 的结果返回。所以，Shape 已知的一大约束是，其 json.Marshal 结果必须符合 API 层的预期。

至于在“实战二”的代码实现下，我们对 Shape 完整的约束是什么样的，欢迎你留言讨论。

数据结构

明确了使用界面，下一步就要考虑实现相关的内容。可能大家都听过这样一个说法：

程序 = 数据结构 + 算法

它是一个很好的指导思想。所以当我们谈程序的实现时，我们总是从数据结构和算法两个维度去描述它。

我们先看数据结构。

对于服务端程序，数据结构不完全是我们自己能够做主的。在“[36 | 业务状态与存储中间件](#)”这一讲中我们说过，存储即数据结构。所以，服务端程序在数据结构这一点上，最为重要的一件事是选择合适的存储中间件。然后我们再在该存储中间件之上组织我们的数据。

对于 QPaint 的服务端程序来说，我们选择了 mongodb。

为何是 mongodb，而不是某种关系型数据库？

最重要的理由，是因为图形（Shape）对象的开放性。因为图形的种类很多，它的 Schema 不是我们今天所能够提前预期的。故此，文档型数据库更为合适。

确定了基于 mongodb 这个存储中间件，我们下一步就是定义表结构。当然表（Table）是在关系型数据库中的说法，在 mongodb 中我们叫集合（Collection）。但是出于惯例，我们很多时候还是以“定义表结构”一词来表达我们想干什么。

我们定义了两个表（Collection）：drawing 和 shape。其中，drawing 表记录所有的 drawing，而 shape 表记录所有的 shape。具体如下：

drawing 表

字段名	含义	索引	类型
_id	DrawingID	唯一索引	string
uid	UserID	索引	UserID
shapes	Shape数组	-	[]ShapeID

shape 表

字段名	含义	索引	类型
dgid	DrawingID	-	string
spid	ShapeID	(dgid, spid) 联合唯一索引	ShapeID
shape	Shape	-	json

我们重点关注索引的设计。

在 drawing 表中，我们为 uid 建立了索引。这个比较容易理解：虽然目前我们没有提供 List 某个用户所有 drawing 的方法，但这是迟早的事情。

在 shape 表中，我们为 (dgid, spid) 建立了联合唯一索引。这是因为 spid 作为 ShapeID，是 drawing 内部唯一的，而不是全局唯一的。所以，它需要联合 dgid 作为唯一索引。

算法

谈清楚了数据结构，我们接着聊算法。

在“程序 = 数据结构 + 算法”这个说法中，“算法”指的是什么？

在架构过程中，需求分析阶段，我们关注用户需求的精确表述，我们会引入角色，也就是系统的各类参与方，以及角色间的交互方式，也就是用户故事。


到了详细设计阶段，角色和用户故事就变成了子系统、模块、类或者函数的使用界面（接口）。我们前面一直在强调，使用界面（接口）应该自然体现业务需求，就是强调程序是为

用户需求服务的。而我们的架构设计，在需求分析与后续的概要设计、详细设计等过程之间也有自然的延续性。

所以算法，最直白的含义，指的是用户故事背后的实现机制。


数据结构 + 算法，是为了满足最初的角色与用户故事定义，这是架构的详细设计阶段核心关注点。以下是一些典型的用户故事：

创建新 drawing (uid):

 复制代码


```
1 dgid = newObjectId()
2 db.drawing.insert({_id: dgid, uid: uid, shapes: []})
3 return dgid
```

取得 drawing 的内容 (uid, dgid):

 复制代码

```
1 doc = db.drawing.findOne({_id: dgid, uid: uid})
2 shapes = []
3 foreach spid in doc.shapes {
4     o = db.shape.findOne({dgid: dgid, spid: spid})
5     shapes.push(o.shape)
6 }
7 return shapes
```

删除 drawing (uid, dgid):

 复制代码

```
1 if db.drawing.remove({_id: dgid, uid: uid}) { // 确保用户可删除该 drawing
2     db.shape.remove({dgid: dgid})
3 }
```

创建新 shape (uid, dgid, shape):

```
1 if db.drawing.find({_id: dgid, uid: uid}) { // 确保用户可以操作该 drawing
2     db.shape.insert({dgid: dgid, spid: shape.id, shape: shape})
3     db.drawing.update({$push: {shapes: shape.id}})
4 }
```

删除 shape (uid, dgid, spid):

```
1 if db.drawing.find({_id: dgid, uid: uid}) { // 确保用户可以操作该 drawing
2     if db.drawing.update({$pull: {shapes: spid}}) {
3         db.shape.remove({dgid: dgid, spid: spid})
4     }
5 }
```

这些算法的表达整体是一种伪代码。但它也不完全是伪代码。如果大家用过 mongo 的 shell 的话，其实能够知道这里面的每一条 mongo 数据库操作的代码都是真实有效的。

另外，从严谨的角度来说，以上算法中凡是涉及到多次修改操作的，都应该以事务形式来做。比如删除 drawing 的代码：

```
1 if db.drawing.remove({_id: dgid, uid: uid}) { // 确保用户可删除该 drawing
2     db.shape.remove({dgid: dgid})
3 }
```

假如第一句 drawing 表的 remove 操作执行成功，但是在此时发生了故障停机事件导致 shape 表的 remove 没有完成，那么从用户的业务逻辑角度来说一切都正常，但是从系统维护的角度来说，系统残留了一些孤立的 shape 对象，永远都没有机会被清除。

网络协议

考虑到底层的业务逻辑实现层已经支持多租户，我们网络协议也需要做出相应的修改。这一讲我们只做最简单的调整，引入一个 mock 的授权机制。如下：


```
1 Authorization QPaintStub <uid>
```

既然有了 Authorization，那么我们就不能继续用 restrpc.Env 作为 RPC 请求的环境了。我们自己实现一个 Env，如下：

```
1 type Env struct {
2     restrpc.Env
3     UID UserID
4 }
5
6 func (p *Env) OpenEnv(rcvr interface{}, w *http.ResponseWriter, req *http.Request) error {
7     auth := req.Header.Get("Authorization")
8     pos := strings.Index(auth, " ")
9     if pos < 0 || auth[:pos] != "QPaintStub" {
10         return errBadToken
11     }
12     uid, err := strconv.Atoi(auth[pos+1:])
13     if err != nil {
14         return errBadToken
15     }
16     p.UID = UserID(uid)
17     return p.Env.OpenEnv(rcvr, w, req)
18 }
```

把所有的 restrpc.Env 替换为我们自己的 Env，再对代码进行一些微调（Document 类的调用增加 env.UID 参数），我们就完成了基本的多租户改造。

改造后完整的 RESTful API 层代码如下：

<https://github.com/qiniu/qpaint/blob/v42/paintdom/service.go>

结语

总结一下今天的内容。

今天我们主要改造的是底层的业务逻辑实现层。

一方面，我们对使用界面（接口）作了多租户的改造。多租户改造从网络协议角度来说，主要是增加授权（Authorization）。从底层的 DOM 接口角度来说，主要是 Document 类增加 uid 参数。

另一方面，我们基于 mongodb 完成了新的实现。我们对数据结构和算法作了详细的描述。要更完整了解实现细节，请重点阅读以下两个文件：

https://github.com/qiniu/qpaint/blob/v42/paintdom/README_IMPL.md

<https://github.com/qiniu/qpaint/blob/v42/paintdom/drawing.go>

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 41 | 实战（一）：“画图”程序后端实战

下一篇 43 | 实战（三）：“画图”程序后端实战

精选留言 (5)

写留言



侯永强

2019-09-17

这个专栏看到现在的体会:

- 1.作者是大神。神一样的思考和行为。
- 2.体会了看懂后的快乐，无价之宝。
- 3.必须磕头才能表达敬意。

展开 ▾



3



Aaron Cheung

2019-09-17

读了好几遍文章 很有收获 打卡

展开 ▾



1



Geek_88604f

2019-09-22

关于Sharp的约束我试着理解一下，语言方面的约束老师已经说了两点:一是Drawing类通过接口来引用Sharp类以增加通用性；二是考虑到 Drawing 类的 List 和 Get 返回的 Shape 实例，会被直接作为 RESTful API 的结果返回。所以，Shape 的 json.Marshal 结果必须符合 API 层的预期。

除了语言方面的约束外还存在语意方面的约束，一方面RESTful要求对资源的操作只...
展开 ▾

作者回复: Shape对象的约束和网络没有关系



Geek_88604f

2019-09-22

多租户共用一颗 DOM 树，会不会存在性能瓶颈？在后续用户数增多，高并发的情况下无法满足要求，存在拆分的诉求？

作者回复: 不是共享dom树，这里和共享没什么关系。你可以认为uid和任何普通的变量一样，只是一个普通数据而已。





吴

2019-09-17

有深度

展开 ∨

