

## 63 | 接口设计的准则

2019-12-10 许式伟

许式伟的架构课

[进入课程 >](#)



讲述：姚迪迈

时长 09:21 大小 8.57M



你好，我是七牛云许式伟。

上一讲 “[62 | 重新认识开闭原则 \(OCP\)](#)” 我们介绍了开闭原则。这一讲的内容非常非常重要，可以说是整个架构课的灵魂。总结来说，开闭原则包含以下两层含义：

第一，模块的业务要稳定。模块的业务遵循“只读”设计，如果需要变化不如把它归档，放弃掉。这种模块业务只读的思想，是架构治理的基础哲学。我平常和小伙伴探讨模块边界的时候，经常会说这样一句话：

每一个模块都应该是可完成的。

这实际上是开闭原则的业务范畴 “只读” 的架构治理思想的另一种表述方式。

第二，模块业务的变化点，简单一点的，通过回调函数或者接口开放出去，交给其他的业务模块。复杂一点的，通过引入插件机制把系统分解为 “最小化的核心系统 + 多个彼此正交的周边系统”。事实上回调函数或者接口本质上就是一种事件监听机制，所以它是插件机制的特例。

今天，我们想聊聊怎么做接口设计。

不过在探讨这个问题前，我想和大家探讨的第一个问题是：什么是接口？

你可能会觉得这个问题挺愚蠢的。毕竟这几乎是我们嘴巴里天天会提及的术语，会不知道？但让我们用科学家的严谨作风来看待这个问题。接口在不同的语义环境下，主要有两个不同含义。

一种是模块的使用界面，也就是规格，比如公开的类或函数的原型。我们前面在这个架构课中一直强调，模块的接口应该自然体现业务需求。这里的接口，指的就是模块的使用界面。

另一种是模块对依赖环境的抽象。这种情况下，接口是模块与模块之间的契约。在架构设计中我们经常也会听到 “契约式设计 (Design by Contract)” 这样的说法，它鼓励模块与模块的交互基于接口作为契约，而不是依赖于具体实现。

对于这两类的接口语义，我们分别进行讨论。

## 模块的使用界面

对于模块的使用界面，最重要的是 KISS 原则，让人一眼就明白这个模块在做什么样的业务。

KISS 的全称是 Keep it Simple, Stupid，直译是简单化与傻瓜化。用土话来说，就是要 “让傻子也能够看得懂”，追求简单自然，符合惯例。

这样说比较抽象，我们拿七牛开源的 mockhttp 项目作为例子进行说明。

这个项目早期的项目地址为：

代码主页: <https://github.com/qiniu/mockhttp.v1>

文档主页: <https://godoc.org/github.com/qiniu/mockhttp.v1>

最新的项目地址变更为:


代码主页: <https://github.com/qiniu/x/tree/master/mockhttp>

文档主页: <https://godoc.org/github.com/qiniu/x/mockhttp>

mockhttp 是做什么的呢? 它用于启动 HTTP 服务作为测试用途。


当然 Go 的标准库 [net/http/httptest](https://golang.org/pkg/net/http/httptest/) 已经有自己的 HTTP 服务启动方法, 如下:

```
1 package httptest
2
3 type Server struct {
4     URL string
5     ...
6 }
7
8 func NewServer(service http.Handler) (ts *Server)
9 func (ts *Server) Close()
```


 复制代码

httptest.NewServer 分配一个空闲可用的 TCP 端口, 并将它与传入的 HTTP 服务器关联起来。最后我们得到的 ts.URL 就是服务器的访问地址。使用样例如下:

```
1 import "net/http"
2 import "net/http/httptest"
3
4 func TestXXX(t *testing.T) {
5     service := ... // HTTP 业务服务器
6     ts := httptest.NewServer(service)
7     defer ts.Close()
8
9     resp, err := http.Get(ts.URL + "/foo/bar")
10    ...
11 }
```


 复制代码

mockhttp 有所不同，它并不真的启动 HTTP 服务，没有端口占用。这里我们不谈具体的原理，我们看接口。mockhttp.v1 版本的使用界面如下：

 复制代码

```
1 package mockhttp
2
3 var Client rpc.Client
4
5 func Bind(host string, service interface{})
```

这里比较古怪的是 service，它并不是 http.Handler 类型。它背后做了一件事情，就是帮 service 这个 HTTP 服务器自动实现请求的路由分派能力。这有一定的好处，使用上比较便捷：

 复制代码

```
1 import "github.com/qiniu/mockhttp.v1"
2
3 func TestXXX(t *testing.T) {
4     service := ... // HTTP 业务服务器
5     mockhttp.Bind("example.com", service)
6     resp, err := mockhttp.Client.Get("http://example.com/foo/bar")
7     ...
8 }
```

但是它有两个问题。


一个问题是关于模块边界上的。严谨来说 mockhttp.v1 并不符合 “单一职责原则 (SRP) ”。它干了两个业务：

启动 HTTP 测试服务；

实现 HTTP 服务器请求的路由分派。


另一个是关于接口的 KISS 原则。mockhttp.Bind 虽然听起来不错，也很简单，但实际上并不符合 Go 语言的惯例语义。另外就是 mockhttp.Client 变量。按 Go 语义的惯例它可能叫 DefaultClient 会更好一些，另外它的类型是 rpc.Client，而不是 http.Client，这样方便是方便了，但却产生了多余的依赖。

mockhttp.v1 这种业务边界和接口的随意性，一定程度上是因为它是测试用途，所以有点怎么简单怎么来的意思。但是后来的发展表明，所有的偷懒总会还回来的。于是就有了 mockhttp.v2 版本。这个版本在我们做小型的 package 合并时，把它放到了 <https://github.com/qiniu/x> 这个 package 中。接口如下：

 复制代码

```
1 package mockhttp
2
3 var DefaultTransport *Transport
4 var DefaultClient *http.Client
5
6 func ListenAndServe(host string, service http.Handler)
```

这里暴露的方法和变量，一方面 Go 程序员一看即明其义，另一方面语义上和 Go 标准库既有的 HTTP package 可自然融合。它的使用方式如下：

 复制代码

```
1 import "github.com/qiniu/x/mockhttp"
2
3 func TestXXX(t *testing.T) {
4     service := ... // HTTP 业务服务器
5     mockhttp.ListenAndServe("example.com", service)
6     resp, err := mockhttp.DefaultClient.Get("http://example.com/foo/bar")
7     ...
8 }
```

从上面的例子可以看出，我们说接口要 KISS，要简单自然，这里很重要的一点是符合语言 and 社区的惯例。如果某类业务在语言中已经有约定俗成的接口，我们尽可能沿用相同的接口语义。

## 模块的环境依赖

接口的另一种含义是模块对依赖环境的抽象，也就是模块与模块之间的契约。我们大部分情况下提到的接口，指的是这一点。

模块的环境依赖，也分两种，一种是使用界面依赖，一种是实现依赖。所谓使用界面依赖是指用户在使用该模块的使用界面时自然涉及的。所谓实现依赖则是指模块当前实现方案中涉

及到的组件，它带来的依赖条件。如果我换一种实现方案，这类依赖可能就不再存在，或者变成另外的依赖。

在环境依赖上，我们遵循的是“最小依赖原则”，或者叫“最少知识原则（Least Knowledge Principle, LKP）”，去尽可能发现模块中多余的依赖。

具体到细节，使用界面依赖与实现依赖到处置方式往往还是有所不同。

从使用界面依赖来说，我们接口定义更多考虑的往往是对参数的泛化与抽象，以便让我们可以适应更广泛的场景。

比如，我们前面谈到 IO 系统的时候，把存盘与读盘的接口从 `*.os.File` 换成 `io.Reader`、`io.Writer`，以获得更强的通用性，比如对剪贴板的支持。

类似的情况还有很多，一个接口的参数类型稍加变化，就会获得更大的通用性。再比如，对于上面 `mockhttp.v1` 中 `rpc.Client` 这个接口就存在多余的依赖，改为 `http.Client` 会更好一些。

不过有的时候，我们看起来从接口定义似乎更加泛化，但是实际上却是场景的收紧，这需要特别注意避免的。比如上面 `mockhttp.v1` 的接口：

```
1 func Bind(host string, service interface{})
```

 复制代码

与 `mockhttp.v2` 的接口：

```
1 func ListenAndServe(host string, service http.Handler)
```

 复制代码

看似 `v1` 版本类型用的是 `interface{}`，形式上更加泛化，但实际上 `v1` 版本有更强的假设，它内部通过反射机制实现了 HTTP 服务器请求的路由分派。而 `v2` 版本对 `service` 则用的是 HTTP 服务器的通用接口，是更加恰如其分的描述方式。

当然，在接口参数的抽象上，也不适合过度。如果某种泛化它不会发生，那就是过度设计。不要一开始就把系统设计得非常复杂，而陷入“过度设计”的深渊。应该让系统足够的简单，而却又不失扩展性，这其中的平衡完全依赖你对业务的理解，它是一个难点。

聊完使用界面依赖，我们接着聊实现依赖。

从模块实现的角度，我们环境依赖有两个选择：一个是直接依赖所基于的组件，一个是将所依赖的组件所有被引用的方法抽象成一个接口，让模块依赖接口而不是具体的组件。

那么，这两种方式应该怎么选择？

我的建议是，大部分情况下应该选择直接依赖组件，而不必去抽象它。

如无必要，勿增实体。

如果我们大量抽象所依赖的基础组件，意味着我们系统的可配置性（Configurable）更好，但学习成本也更高。

什么时候该当考虑把依赖抽象化？

其一，在需要提供多种选择的时候。比较典型的是日志的 Logger 组件。对于绝大部分的业务模块，都并不希望绑定 Logger 的选择，把决策权交给使用方。

但是有的时候，在这一点上过度设计也会比较常见。比如，不少业务模块会选择抽象对数据库的依赖，以便于在 MySQL 和 MongoDB 之间自由切换。但这种灵活性绝大部分情况下是一种过度设计。选择数据库应该是非常谨慎严谨的行为。

其二，在需要解除一个庞大的外部系统的依赖时。有时候我们并不是需要多个选择，而是某个外部依赖过重，我们测试或其他场景可能会选择 mock 一个外部依赖，以便降低测试系统的依赖。

其三，在依赖的外部系统为可选组件时。这个时候模块会实现一个 mock 的组件，并在初始化时将接口设置为 mock 组件。这样的好处是，除非用户关心，否则客户可以当模块不存在这个可选的配置项，这降低了学习门槛。

整体来说，对模块的实现依赖进行接口抽象，本质是对模块进行配置化，增加很多配置选项，这样的配置化需要谨慎，适可而止。

## 结语

接口设计是一个老生常谈的话题。接口有分模块的使用界面和模块的环境依赖这两种理解。

对于模块的使用界面，我们推崇 KISS 原则，简单自然，符合业务表达的惯例。

对于模块的环境依赖，我们遵循的是 “最小依赖原则”，或者叫 “最少知识原则 (Least Knowledge Principle, LKP) ”，尽可能发现模块中多余的依赖。

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。下一讲我们的话题是 “不断完善的架构范式”。

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。

# 21 天打卡行动

- 99 元报名参与打卡
- 连续坚持 21 天
- 全额退还报名费

报名即赠 ¥199 奖金 

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。



## 精选留言 (9)

写留言



Aaron Cheung

2019-12-10

所以orm是否还有必要呢 ruby python go 都有挺多ORM

作者回复: 少用



2



吴

2019-12-11

老师，是我们的平台想记录用户的浏览轨迹了，我又不想将记录功能分布到各个模块中，有什么好的办法吗？

作者回复: 在入口去记录。比如服务端的入口进行记录，或者前端统一进行记录。

1

1



吴

2019-12-11

老师，浏览日志和操作日志怎么设计合理一些

展开

作者回复: 就是文中说的 logger？一般在应用最外层选定 logger，可能写到一个滚动的日志文件，也可能发到一个分布式的日志收集平台。

1

1



Jxin

2019-12-10

mock外部依赖，以实现本服务的独立测试与交付。

展开



1



**靠人品去赢**

2019-12-10

接口其实就是一个解耦的，你别管我怎么实现的你就按着接口来传参就好了。  
所以我觉得，所以内部其实可以少用接口，继承也要更少用，多用组合的方式。  
对外部提供接口，尽量的设计好，不要一大堆参数，实在不行你传个对象也行，保证最简洁

展开 ∨



1



**Charles**

2019-12-10

架构师应该站在全局高位考虑项目，所以开发效率和架构设计以及扩展之间，有时候追求的是一种平衡，没绝对是吗？

展开 ∨



**leslie**

2019-12-10

“大部分情况下应该选择直接依赖组件，而不必去抽象”说起这个其实就像我们去提及工具或者说功能和可扩展性的取舍。用中间件存储的rabbitMQ和kafka在高并发方面来举例：

rabbitMQ:rabbit在高并发场景下确实比kafka强-阿里多次双11中历经考验，不过源代码的空间改造性相对kafka难许多，符合老师所说的直接用，不过一旦使用其替代方...

展开 ∨



**有铭**

2019-12-10

我记得ORM这个东西之所以诞生的一个重要原因就是大约15年，切换关系数据库是一种刚需，当然现在已经是伪命题了

展开 ∨



**丁丁历险记**

2019-12-10

目读完后想听一遍，发现没有声音，睡觉。

展开 ∨



