



下载APP



大咖助阵 | 大明：抽象，抽象，还是抽象

2021-11-24 大明

《手把手带你写一个Web框架》

课程介绍 >

**讲述：正霖**

时长 18:09 大小 16.64M



你好，我是大明。

之前我在极客时间的毛剑老师的 Go 进阶训练营里，为基础不太好的同学开了一个小课，主要是通过讲如何设计一个 Web 框架来带领学员一起巩固 Go 基础，侧重点在 Go 语法上，没有深入讨论设计本身以及设计理念。

但是，教会如何用一个框架是个很简单的问题，教会如何做抽象才是一个很难、但很有价值的问题。所以今天我想借这篇加餐的机会，围绕 Web 框架的设计，来分享一下我平时在做设计和落地时对抽象的一些体会。



在开始讨论之前，我们要先搞清楚一个问题：为什么抽象那么重要？

因为我认为，**程序是人对现实世界的抽象的形式化描述**。它包含两层含义：第一个含义是要先建立对现实世界抽象；第二个是能够将这个抽象落地为代码，也就是设计。因此，抽象是设计的前提。

通俗点就是，如果你不能正确认识问题，怎么奢求自己能够正确解决问题呢？所以，我们先稍微讨论一下 Web 框架，看看它背后对应的问题。

Web 框架

Web 框架要解决的问题是什么？处理 HTTP 请求。进一步细想，我们会想到具体如何处理请求，但这不是 Web 框架的事情，而是用户的事情。比如说用户登录的请求，Web 框架肯定是不知道如何处理的。

所以我们能够认识到，**Web 框架主要负责：接收请求、找到用户处理逻辑调用一下、再把响应写回去。也就是三件事：接受请求、分发请求、写回响应**。如果用伪代码来描述就是：

```
1 while:
2     req = next()
3     handler = findHandler(req)
4     response = handler.Handle(req)
5     write(respons)
```

[复制代码](#)

其中 Go 的 http/net 包帮我们解决了读取请求和写回输出的大部分工作，只剩下路由部分，即根据请求找到用户注册的处理方法，需要我们支持一下。这就是 Web 框架的核心，同时也是最难的地方。

但是你去看市面上大多数 Web 框架不仅仅只有这个基础功能，还有很多花里胡哨的东西。大致列举一下：

参数解析校验功能

文件操作类，比如说上传、下载，或者作为静态资源服务器

特定格式支持，比如说 JSON、XML 格式支持

模板支持，主要是用于渲染页面

AOP 解决方案

...

一个框架一般不会对初期把所有都支持好，而是在逐步迭代的过程中，将功能补充上去。所以要求我们在设计框架之初，就要考虑到这一类的功能。但是并不需要提供实现，甚至连接口都不必设计得一步到位，只要藏好细节不暴露出去，后面可以轻易修改而不影响已有的用户。

所以简单总结下，一个 Web 框架要解决两大类的问题：

必须要解决的核心问题，在 Web 框架里就是路由问题


可解决可不解决的次级问题，也就是这里罗列的一大堆

问题搞清楚，我们再看怎么解决。其实，构建抽象就是解决问题的过程。

如何构建抽象

我们从前面的伪代码里面，基本上就可以抽取出来第一个也是最重要的抽象：

```
1 type Handler interface {  
2     ServeHTTP(c *Context)  
3     Rountable  
4 }
```

 复制代码

需要注意的是，这是我设计的接口。不同的人来设计，是可以不同的，只要都能表达这么一种抽象就可以。也就是，重要的是抽象，而抽象的表达形式是多样的。

ServeHTTP 精准描述了这个 Web 框架的主要责任，也准确界定了 Web 框架的边界，我称为核心抽象。

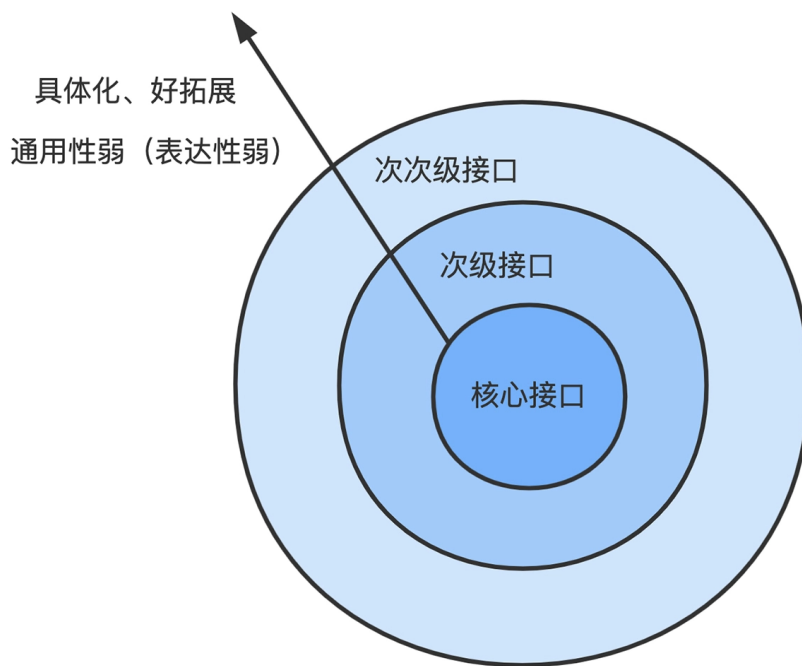
而组合进来的 Rountable 接口实际上只是支撑接口，也就是去掉它并不妨碍我们用 Handler 来表达整个 Web 框架的主要责任。所以我把类似的东西称为次级抽象。

次级抽象并不是描述了整个框架，而是描述了框架的一个方面。次级抽象一般用于支撑核心抽象的实现。显然，如果我们实现次级接口，自然可能会有支撑该实现的次次级接口。

因此整个框架的实现过程可以简化为：

1. 构建抽象，提供实现
2. 为了特定实现，引入新的抽象
3. 为新的抽象提供实现

如此不断迭代，这种设计方式，我一般称为自顶向下的设计，从核心接口的抽象向外延伸。例如在这个 Web 框架里，为了实现 Handler 而引入 Routable，而在实现 Routable 的时候引入 node，它们之间的关系，大体上可以看作是一个同心圆。




极客时间

一般来说，越是接近核心的接口，越是稳定，缺点则是过于抽象以至于对普通用户来说毫无价值。甚至实现一个核心接口，难度差不多相当于重写整个框架了。

对应地，远离核心的接口，其抽象程度低，因此也就更加具体化。更加具体化，意味着好扩展，缺点则是表达性弱，只能表达很小的一个点。注重使用感的最外围的接口，大多数时候是框架设计者有意留出来给用户使用的，实现非常简单。

例如说，为了解决优雅退出，允许用户注册自己的清理动作，我引入了一个新的抽象：

 复制代码

```
1 type Hook func(ctx context.Context) error
```

显而易见地，写一个 Hook 和写一个 Handler 的难度自然是差了好几个数量级。

到这里我们讨论了如何构建抽象，然而不管我们再怎么自信，都还有一个问题要解决，就是抽象不是一成不变的，随着业务发展，之前的抽象就可能不适用了，或者说至少某个部分不再适用了。因此我们总是要在设计的时候考虑变更的问题。

想写出能变更的代码，首先要考虑的就是，到底哪些地方要考虑变更？

如何识别变更点

识别变更，基本上，要么依赖于个人的经验，要么依赖于借鉴类似的框架。

借鉴这个办法很简单，所谓的太阳底下无新鲜事，在我们这一行尤其如此。大多数时候你都可以找到类似的产品，看看它们是如何设计的。这里就要强调一点，解决同一个问题的框架，它们落地为代码之后，都是千差万别的，**所以借鉴，是要借鉴它们的抽象，而不是它们的代码。**

此外就是要借鉴缺点，所谓的每与操反，事乃可成。

例如有的框架内置了处理静态资源的功能，但是该功能和框架的核心路由功能耦合在一起了，缺乏扩展性和可读性，还对用户重构核心功能造成了阻碍。这个设计就不是很好，我们自己设计 Web 框架功能的时候就可以借鉴这一点。我们可以写一个方法，就像用户写的普通业务方法，这样用户可以按类似用法注册一个路由启用这个功能，更方便。

我就重点说通过我多年个人经验总结的两个小技巧：假设法和替换法。

假设法

所谓假设法，就是找到自己实现过程中做的假设。最常见的假设就是，“产品说这个场景不需要考虑，所以我只需要这么做就可以了”，又或者“我觉得这么做就可以了，因为用

户不可能需要 xxx”每当在脑海里面出现这一类的语句，就代表了我们做了一些假设。

但凡假设不成立，就意味着实现要变更。而不管是在中间件设计，还是在业务设计上，我们总是会自觉或者不自觉地引入很多假设。

比如说，在我学习 Web 框架时写代码，做了一个很重要的假设，朝前匹配，这个假设极大简化了实现的代码。例如两个路由：

```
1 /api/user/*/profile
2 /api/user/xiaoming/home
```

[复制代码](#)

如果我们的请求是 `/api/user/xiaoming/profile`，在匹配到 `xiaoming` 的时候，应该走去第二个路由，但是后面的 `profile` 和 `home` 匹配不上。在工业级的 Web 框架里面，显然它们会回溯，找到第一个路由继续匹配。但是这会导致代码非常复杂。

我并不希望在这个以学习为目标的 Web 框架里面引入这种回溯的机制，所以引入了一个假设——路由匹配总是朝前的。显然，这个假设非常脆弱，但凡我想要把这个 Web 框架支持到工业级，首先就要重写这部分代码以正确匹配上第一个路由。

这个方法，**难就难在识别不自觉的假设，特别是一些业务规则的假设**。因为我们太熟悉这些业务规则，以至于默认它就是这样的，都不觉得自己做了假设。

典型的例子是金额，大部分国内开发者在处理金额的时候几乎不会考虑货币的问题，因为我们默认就是人民币，因而接口、数据库都只需要一个数字来表达金额。而一旦业务扩张到外币，就会发现几乎全部接口都缺乏货币的支持，整个改造起来就耗时长范围广了。

替换法

在讲框架设计的时候，我经常被问到一类问题：这个东西是做成这样还是做成那样？这个时候我都会建议他抽取出来一个接口，先写自己更加偏好的做法。将来在需要的时候，再替换为另外一个实现。这就是典型的替换场景。

这一类的场景识别起来也很容易，就是解决方案不唯一。不唯一就代表，今天可能用这个，明天可能用那个。**如果你陷入一种进退两难的境地，又或者识别出这一块业界有多种**

做法，就可以用这个方法。

这里依旧用 Web 框架的例子。我在写路由树的时候发现，“某个节点是否匹配某段路径”是一个显而易见的扩展点，也就是方法：

[复制代码](#)

```
1 func (h *HandlerBasedOnTree) findMatchChild(root *node, path string) (*node, b
2     for _, child := range root.children {
3         if child.path == path {
4             return child, true
5         }
6     }
7     return nil, false
8 }
```

这是第一版代码，里面是严格匹配路径的，也就是条件 `child.path==path` 是变更点。为什么这么说呢？因为我使用过别的 Web 框架，知道还可以忽略大小写匹配、通配符匹配、参数路径以及正则匹配之类的匹配方式。

因此，“如何匹配”就是一个变更点。也可以说，“如何匹配”是一个选择匹配策略的问题。

如何适应变更

变更的位置我们找出来了，但是问题来了，我们需要为每一个变更点都设计一套接口吗？或者说，需要立刻解决它吗？

前面金额的例子说到大多数国内开发者并不会有意设计货币。但是在这里即便我识别出来了，如果公司现在没有这一类的业务，我也不会设计货币。因为过早引入货币，会导致前期开发不得不处理这些问题，拖累开发进度。

但是大多数时候，我们识别出来的变更点都是需要立刻处理的，一般来说我采用“隔离”和“超前一步设计”两个技巧。

隔离

隔离是最简单的处理方式，一般是抽取出来作为一个方法，或者一个结构体。

比如说前面的 `findMatchChild` 方法，就是我隔离出来的方法。因为在最开始的版本里，我知道这个地方会变，也就是说匹配规则是多样的，但是那个时候我还不想理它，又害怕完全不理吧，后面变了会波及其它代码，就抽取出来作为一个单独的方法。

那么后续不管怎么变，比如说支持路径参数、正则匹配，它都局限在了这个方法内，至多就是修改一下方法签名，传入更多参数。而对于调用方来说，传入更多参数会有点影响，但是影响已经是非常可控的。

如果抽取得好，遇到变更的时候，完全可以把这个方法升级成为一个接口。实际上，这也是从隔离走向超前一步设计的典型场景。

抽取结构体这种做法，一般是用于非常复杂的变更点。即这个变更点现有的逻辑就足够复杂，将来也很复杂，所以用一个结构体来封装一系列的方法。

但是不管是抽成方法还是结构体，就我个人经验而言，隔离的核心是要做到，**隔离出来的是一个可测试的单元**。什么意思呢？就是这个隔离出来的产物，一定要是单独可测试的。

因为将来变更之后，你可以完全复用当下的测试代码，来测试它变更前后的对外表现出来的行为是否一致。如果不一致，要么是我们的变更有问题，要么就是之前的隔离其实并不充分，这种情况下往往意味着使用方也要大动干戈。

超前一步设计

在前面我就已经暗示了如何做到超前一步设计。

超前一步的精髓在于：**设计接口和接入实现的方式，但是不提供多样化的接口实现**。这句话的意思是说，识别出变更点的时候，我们要设计一个接口出来，但是并不需要为所有可能的变更提供实现，只提供当下需要的实现即可。

设计接口这一步，说简单也简单，只需要参考隔离，把可能变化的逻辑抽取出来作为一个方法，然后这个方法的定义，就是接口内部方法的定义。如果是隔离出来结构体，那这个结构体就可以看作是接口的一个实现了。

如果说难，那么就难在做到这个接口将来肯定不会发生任何变更，也就是方法的输入、输出一点不变，这近乎不可能了。但是可以尝试减轻这种变更的影响，例如说输入和输出都

定义为一个结构体，后续的变更无非就是增加字段而已。

举个例子，在意识到“如何匹配”是一个变更点之后，我们就可以在 node 中引入另外一个抽象 matchFunc：

[复制代码](#)

```
1 type node struct {
2     children []*node
3     // 如果这是叶子节点，
4     // 那么匹配上之后就可以调用该方法
5     handler handlerFunc
6     matchFunc matchFunc
7     // 原始的 pattern。注意，它不是完整的pattern，
8     // 而是匹配到这个节点的pattern
9     pattern string
10    nodeType int
11 }
```

但是，这个时候需要提供所有可能的实现吗？并不需要，我们只需要提供满足当前需求的实现就可以了。例如说，只支持简单的通配符匹配和路径参数匹配，但是不需要支持正则匹配。毕竟正则匹配比较少用，可以推迟到有用户反馈需要的时候再提供。

另外一个难点是，如何设计一个优秀的接入方式。因为这个机制要解答两个问题：知道用哪个实现并且得到对应的实例。如果说，**接口的质量决定了你的设计能不能适应变更，那么接入机制就是决定了你能多快适应变更。**

Go 不同于 Java，没有 SPI，也没有 Spring 这种提供容器的框架，连动态加载包的功能也没有。所以 Go 设计一个好用的接入机制，比较难，手段也比较单一。典型的做法就是 Register + init。

Register 是指设计一个注册实现的方法。例如在 Web 框架里面注册 Filter 的实现：


[复制代码](#)

```
1 var builderMap = make(map[string]FilterBuilder, 4)
2 func RegisterFilter(name string, builder FilterBuilder) {
3     builderMap[name] = builder
4 }
5 func GetFilterBuilder(name string) FilterBuilder {
6     return builderMap[name]
```

7 }

这种是比较复杂的，框架内部通过按名索引来获得实例。如果只需要单一实现，那么可以将 map 去掉，直接存储实例。

Register 要结合 init 方法来注入实现：

 复制代码

```
1 // 匿名引入 _ "package/to/this/filters"
2 func init() {
3     web.RegisterFilter("my-custom", myFilterBuilder)
4 }
```

用户需要在启动的地方，使用匿名引入来注入这个实现。

在理想情况下，接入机制应该被设计为无侵入式的。也就是替换一个新的实现，不需要修改代码。只不过这很难，在 Go 里面尤其难。前面 Filter 的接入方式就是无侵入式的，框架不需要有任何的修改。

而 matchFunc 就没有提供任何无侵入式的扩展手段，也就是说用户无法自定义自己的匹配规则，至多是在得到用户反馈之后，依据用户的需要，设计一种新的节点。即便如此，也需要稍微修改接入代码，这是因为我们创建 node 的方式是不一样的：

 复制代码

```
1 // 静态节点
2 func newStaticNode(path string) *node {}
3 // 通配符 * 节点
4 func newAnyNode() *node {}
```

所以 node 的设计在质量上要比 Filter 更差。

Filter 的抽象和接入机制还体现了另外一个原则：**实现平等原则，接口的所有实现，在地位上是平等的，我怎么接入实现，用户就是怎么接入自己的实现。**有些框架在这方面就做得很差，这些框架无一不是给自己的实现提供了特殊的地位，比如说典型的针对自身提供的实现，做了特殊的处理。

总结

我们基本上没有讨论 Web 框架的实现细节，只是借助 Web 框架讨论了一些我遵循的基本设计原则。而这一切的前提，就是抽象。

但你可能有个疑问，很多时候，我们很少设计这样一个完整的框架，那这些知识还有用吗？

答案是有用的，我们可以用这种思路去分析别的框架的源码。

我的分析思路供你参考。**我会假设自己是这个框架设计者，分析框架要解决什么、核心的抽象会是什么。**在看到核心抽象后，想象自己会如何实现，然后再去看实现，看实现引入了什么接口、揣度这个接口是表达了什么抽象、为什么要引入这个接口——本质上也是分析这个接口要解决什么问题。不断递归，直到搞明白框架的基本设计方式。

这里讨论的这种设计方式，我称之为自顶向下的设计方式，也就是从核心抽象出发。它自然也有很强的局限性，最大的局限性就是，如果你不熟悉问题领域，你是连核心抽象都难构建出来的。

所以这一切，我想用毛主席的一句话来总结：谁是我们的朋友，谁是我们的敌人，是革命的首要问题。

欢迎在留言区分享你的思考。如果你觉得有收获，也欢迎把今天的内容分享给你身边的朋友。我们下节课见。

分享给需要的人，Ta 订阅后你可得 **20 元现金奖励**

 生成海报并分享

 赞 3  提建议

© 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 29 | 周边：框架发布和维护也是重要的一环

下一篇 大咖助阵 | 飞雪无情：十年面试经验忠告，不要被框架所束缚

训练营推荐

Java 学习包免费领 NEW

面试题答案均由大厂工程师整理

阿里、美团等
大厂真题

18 大知识点
专项练习

大厂面试
流程解析

可复用的
面试方法

面试前
要做的准备

精选留言 (1)

写留言



Vincent 认证

2021-11-24

高屋建瓴，对于抽象的价值与过程，又加深了

展开 ∨



👍 1