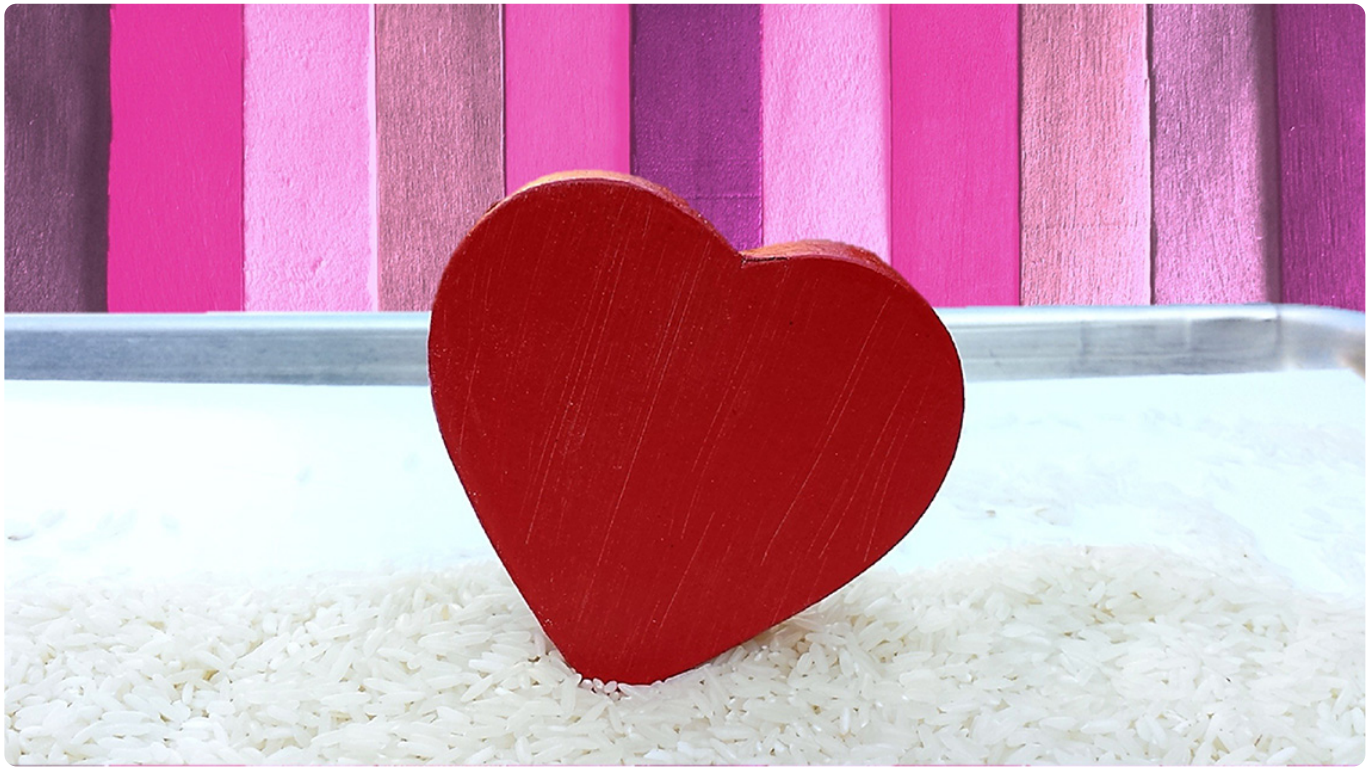
62 | 重新认识开闭原则（OCP）

2019-12-06 许式伟



讲述：姚迪迈

时长 11:13　大小 10.29M

你好，我是七牛云许式伟。

架构的本质是业务的正交分解。

在上一讲 “61 | 全局性功能的架构设计” 中我们提到，架构分解中有两大难题：其一，需求的交织。不同需求混杂在一起，也就是存在所谓的全局性功能。其二，需求的易变。不同客户，不同场景下需求看起来很不一样，场景呈发散趋势。

我们可能经常会听到各种架构思维的原则或模式。但，为什么我们开始谈到架构思维了，也不是从那些耳熟能详的原则或模式谈起？

因为，万变不离其宗。

就架构的本质而言，我们核心要掌握的架构设计的工具其实就只有两个：

组合。用小业务组装出大业务，组装出越来越复杂的系统。

如何应对变化（开闭原则）。

开闭原则（OCP）

今天我们就聊聊怎么应对需求的变化。

谈应对变化，就不能不提著名的 “开闭原则（Open Closed Principle，OCP）” 。一般认为，最早提出开闭原则这一术语的是勃兰特·梅耶（Bertrand Meyer）。他在 1988 年在《面向对象软件构造》 中首次提出了开闭原则。

什么是开闭原则（OCP）？

> 软件实体（模块，类，函数等）应该对于功能扩展是开放的，但对于修改是封闭的。

一个软件产品只要在其生命周期内，都会不断发生变化。变化是一个事实，所以我们需要让软件去适应变化。我们应该在设计时尽量适应这些变化，以提高项目的稳定性和灵活性，真正实现 “拥抱变化” 。

开闭原则告诉我们，应尽量通过扩展软件实体的行为来应对变化，满足新的需求，而不是通过修改现有代码来完成变化，它是为软件实体的未来事件而制定的对现行开发设计进行约束的一个原则。

为什么会有这样的架构设计原则？它背后体现的架构哲学是什么？

本质上，开闭原则的背后，是推崇模块业务的确定性。我们可以修改模块代码的缺陷（Bug），但不要去随意调整模块的业务范畴，增加功能或减少功能都并不鼓励。这意味着，它认为模块的业务变更是需要极其谨慎的，需要经得起推敲的。

我个人非常推崇 “开闭原则” 。它背后隐含的架构哲学，和我说的 “架构的本质是业务的正交分解” 一脉相承。

与其修改模块的业务，不如实现一个新业务。只要业务的分解一直被正确执行的话，实现一个新的业务模块来完成新的业务范畴，是一件极其轻松的事情。从这个角度来说，开闭原则鼓励写 “只读” 的业务模块，一经设计就不可修改，如果要修改业务就直接废弃它，转而实现新的业务模块。

这种 “只读” 思想，大家可能很熟悉。比如基于 Git 的源代码版本管理、基于容器的服务治理都是通过 “只读” 设计来改善系统的治理难度。

对于架构设计来说同样如此。 “只读” 的架构分解让我们逐步沉淀下越来越多可复用的业务模块。如此，我们不断坚持下去，随着时间沉淀，我们的组织就会变得很强大，组装复杂业务系统也将变得越来越简单。

所以开闭原则，是架构治理的根本哲学。

CPU 背后的架构思维

一种广泛的误解认为，开闭原则是一种 “面向对象编程（OOP）” 领域提出来的编程思想。但这种理解显然太过狭隘。虽然开闭原则的正式提出可能较晚，但是在信息科技的发展历程中，开闭原则思想的应用就太多了，它是信息技术架构的基本原则。注意我这里没有用 “软件架构” 而是用 “信息技术架构” ，因为它并不只适用于软件设计的范畴。

我们在 “02 | 大厦基石：无生有，有生万物” 一讲介绍冯·诺依曼体系的规格时就讲过：

> 从需求分析角度来说，关键要抓住需求的稳定点和变化点。需求的稳定点，往往是系统的核心价值点；而需求的变化点，则往往需要相应去做开放性设计。

冯·诺依曼体系的中央处理器（CPU）的设计完美体现了 “开闭原则” 的架构思想。它表现在：

> 指令是稳定的，但指令序列是变化的，只有这样计算机才能够实现 “解决一切可以用‘计算’ 来解决的问题” 这个目标。

> 计算是稳定的，但数据交换是多变的，只有这样才能够让计算机不必修改基础架构却可以适应不断发展变化的交互技术革命。

体会一下：我们怎么做到支持多变的指令序列的？我们由此发明了软件。我们怎么做到支持多变的输入输出设备的？我们定义了输入输出规范。

我们不必去修改 CPU，但是我们却支持了如此多姿多彩的信息世界。

多么优雅的设计。它与面向对象无关，完全是开闭原则带来的威力。

CPU 的优雅设计远不止于此。在 “07 | 软件运行机制及内存管理” 这一讲中，我们介绍了 CPU 对虚拟内存的支持。通过引入缺页中断，CPU 将自身与多变的外置存储设备，以及多变的文件系统格式进行了解耦。

中断机制，我们可以简单把它理解为 CPU 引入的回调函数。通过中断，CPU 把对计算机外设的演进能力交给了操作系统。这是开闭原则的鲜活案例。

插件机制

一些人对开闭原则的错误解读，认为开闭原则不鼓励修改软件的源代码来响应新需求。

这个说法当然有点极端化。开闭原则关注的焦点是模块，并不是最终形成的软件。模块应该坚持自己的业务不变，这是开闭原则所鼓励的。

当然软件也是一个业务系统，但对软件系统这个大模块来说，如果我们坚持它的业务范畴不变，就意味着我们放弃进步。

让软件的代码不变，但业务范畴却能够适应需求变化，有没有可能？

有这个可能性，这就是插件机制。

常规我们理解的插件，通常以动态库（dll/so）形式存在，这种插件机制是操作系统引入的，可以做到跨语言。当然部分语言，比如 Java，它有自己的插件机制，以 jar 包的形式存在。

在上一讲 “61 | 全局性功能的架构设计” 中我们提到，微软的大部分软件，以 Office 和 Visual Studio 为代表，都提供了二次开发能力。

这些二次开发接口构成了软件的插件机制，并最终让它成为一个生态型软件。

一般来说，提供插件机制的二次开发接口需要包含以下三个部分。

其一，软件自身能力的暴露，也就是我们经常说的 DOM API。插件以此来调用软件已经实现的功能，这是最基础的部分，我们这里不进一步展开。

其二，插件加载机制。通常，这基于文件系统，比如我们规定把所有插件放到某个目录下。在 Windows 平台下会多一个选择，把插件信息写到注册表。

其三，事件监听。这是关键，也是难点所在。没有事件，插件没有机会介入到业务中去。但是应该提供什么样的事件，提供多少个事件，这非常依赖架构能力。

原则来说，在提供的能力相同的情况下，事件当然越少越好。但是怎么做到少而精，这非常有讲究。一般来说，事件分以下三类：

其一，界面操作类。最原始的是鼠标和键盘操作，但它们太过于底层，提供出去会是双刃剑，一般对二次开发接口来说会选择不提供。更多的时候会选择暴露更高级的界面事件，比如菜单项或按钮的点击。

其二，数据变更类。在数据发生变化的时候，允许捕获它并做点什么。最为典型的是 onSelectionChanged 这个事件，基本上所有的软件二次开发接口都会提供。当然它属于界面数据变更，只能说是数据变更的特例。如果我们回忆一下 MVC 框架（参见 “22 | 桌面程序的架构建议”），就能够记得 Model 层会发出数据变更通知，也就是 onDataChanged 类的事件出来给 View 或 Controller。

其三，业务流程类。它通常发生在某个业务流的中间某个环节，或者业务流完成之后。比如对 Office 软件来说，打开文件之初或之后，都可能发出相应的事件，以便插件做些什么。

通过以上分析可以看出，完整的插件机制还是比较庞大的。但实际应用中插件机制未必要做得如此之重。

比如，Go 语言中的 image 包，它提供的 Decode 和 DecodeConfig 等功能都支持插件，我们可以增加一种格式支持，而无需修改 image 包。

这里面最大的简化，是放弃了插件加载机制。我们自己手工来加载插件，比如：

```
import "image"
import _ "image/jpeg"
import _ "image/png"
```

这段代码为 image 包加载了两个插件，一个支持 jpeg，一个支持 png 格式。

如果大家仔细研究过我们实战案例 “画图程序” 的代码（参见 “加餐 | 实战：画图程序的整体架构”）就会发现，类似的插件机制的运用有很多。我们说的架构分解，把复杂系统分解为一个最小化的核心系统，加上多个相互正交的周边系统，它背后的机制往往就是我们这里提的插件机制。

插件机制的确让核心系统与周边系统耦合度大大降低。但插件机制并非没有成本。插件机制本身也是核心系统的一个功能，它本身也需要考虑与核心系统其他功能的耦合度。

如果某插件机制没有多少客户，也就是说，没有几个功能基于它开发，而它本身代码又散落在核心系统的各个角落，那么投入产出就显然不成比例。

所以维持足够的通用性，是提供插件机制的重大前提。

单一职责原则

到此为止，相信大家已经对开闭原则（OCP）非常了解了。总结来说就两点：

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。

第二，模块的业务变化点，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。

平常，我们大家也经常会听到 “单一职责原则（Single Responsibility Principle，SRP）”，它强调的是每个模块只负责一个业务，而不是同时干多个业务。而开闭原则强调的是把模块业务的变化点抽离出来，包给其他的模块。它们谈的本质上是同一个问题的两个面。

结语

从来没有人这样去谈架构的本质，也没有人这样解读开闭原则（OCP），对吧?

其实对于这部 “架构课” 的革命性，我自己从没怀疑过。它的内容是精心设计的，为此我准备了十几年。我们用了四章内容来谈信息科技的需求与架构的演进，然后才进入正题。

用写文章的角度来说，这个伏笔的确够深的。

当然这不完全是伏笔。如果我们把整个信息科技看作最大的一个业务系统，我们有无数人在为之努力奋进，迭代它的架构。大家在竟合中形成自然的分工。学习信息科技的演进史，是学习架构的必要组成部分。我们一方面从中学习怎么做需求分析，另一方面也从中体悟做架构的思维哲学。

当然，还有最重要的一点是，我们要知道演进的结果，也就是信息科技最终形成的基础架构。

作为架构师，我们除了做业务架构，还有一个同等难度的大事，就是选择合适的基础架构。基础架构 + 业务架构，才是你设计的软件的全部。作为架构师，千万不要一叶障目，不见泰山，忘记基础架构选择的重要性。

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 “接口设计的准则”。

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。

上一篇　61 | 全局性功能的架构设计

精选留言 (14)

写留言

梦醒十分
2019-12-06

入木三分，好文章!

展开

6

leslie
2019-12-06

老师的课程中提及的两方面是我觉得自己理解的最不好的：一方面"基础架构 + 业务架构，才是你设计的软件的全部"，另一方面"一方面从中学习怎么做需求分析，另一方面也从中体悟做架构的思维哲学"。

如同老师所说的"架构课 的革命性，我自己从没怀疑过。它的内容是精心设计的，为此我准备了十几年",学习的过程中我其实同样在整体梳理自己作为DBA&&OPS十余年松散...

展开

作者回复: 作为架构师，要深刻认识到一点，光理解架构哲学不足以做好架构，它是让我们判断对与错的法则。要真正做好架构，需要做好业务需求分析，做好正交分解，做好模块边界定义。这些不断梳理清楚的“只读”的业务模块，是架构师真正的武器库。

6

K战神
2019-12-06

许大，希望出书。买来收藏。

时不时枕边翻阅体会大佬的思想。

多年以后，庆幸自己这段时间跟着许大的专栏，有了新的想法和思想。

展开

作者回复: 后面应该会出书

3

Jxin
2019-12-06

笔记：

将开闭原则上移到业务系统。业务对外只读，意味着不可变，但不变的业务生命周期是很短暂的，所以要可扩。要扩展还要不变，就倒逼着要做兼容，而兼容可能会导致现有的功能职责不单一，这又倒逼着要对现有的功能做再抽象，以适应更广的“单一职责”。...

展开

作者回复:

2

Yayu
2019-12-06

如何理解“只读”模块？

展开

作者回复: 不是代码只读，是业务范畴只读，接口尽可能只读（增加方法以兼容的方式进行）。

2 1

沫沫（美丽人生）
2019-12-06

许老师好，很感谢您上次关于PaSS问题的回复，读您的文章让我受益匪浅。还想请教一个问题，像Salesforce是基于元数据来构建系统的，元数据在信息架构里属于什么范畴呢，可以展开来讲一讲吗？盼复。

作者回复: Model 层

2 1

tt
2019-12-08

本课感受最深的一句话：

“第二，模块的业务变化点，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...

展开

作者回复: 业务系统都是相通的

阿西吧
2019-12-06

没有基本功，是很难读懂许老师的深层含义打，只能用苦功，下巧劲

吴
2019-12-06

醍醐灌顶，再多的语言也无法表达，最好的架构课

展开

nanquanmama
2019-12-06

值回票价

展开

靠人品去赢

2019-12-06

这么一想，微服务其实也是单一职责原则的实现。像普通业务的话，不清晰以后的方向，可能现在是工具类，后来又搞商城类，侧重点变化可能无法一下子就确定那些是稳定点，做着做着又是一大坨，哪里不行搞哪里，这种怎么解？

展开

作者回复: 需求洞察本来就是架构的难点

Aaron Cheung

2019-12-06

基础架构 + 业务架构，才是你设计的软件的全部。作为架构师，千万不要一叶障目，不见泰山，忘记基础架构选择的重要性。 基础架构是地基

展开

丁丁历险记

2019-12-06

笔记 ocp 是根基，lsv原则用于保障ocp。 srp 让实现lsv 更简单，就一个引起变更职责代码不容易腐化。 这样出来的代码正交性更好，更容易组合。

展开

:)

2019-12-06

通过老师的文章的一些思考。

1. 事情变得复杂往往有两个方面的原因。1.联系耦合过多 2.变化过多
2. 架构的目的就是让事情变得，简单，清晰。
3. 应对耦合过多，可以使用单一原则。
4. 应对变化过多，可以用开闭原则，而其中的"只读"，让自己有种醍醐灌顶的感觉。

展开