61 | 全局性功能的架构设计

2019-12-03 许式伟

讲述：姚迪迈
时长 10:35 大小 9.71M

你好，我是七牛云许式伟。

在上一讲 “加餐 | 实战：画图程序的整体架构” 中，我们结合前面几章的实战案例画图程序来实际探讨我们架构思维的运用。这一篇虽然以加餐名义体现，但它对理解怎么开展和评估架构工作非常关键。

在架构设计中，我们会有一些难啃的骨头。其中最为典型的，就是全局性功能。全局性功能的特征是很难被剥离出来成为独立模块的。我们仍然以大家熟悉的 Office 软件作为例子：

> 读盘 / 存盘：每增加一个功能，都需要考虑这个功能的数据如何存储到磁盘，如何从磁盘中恢复。

Undo/Redo：每增加一个功能，都需要考虑这个功能如何回滚 / 重做，很难剥离。

宏录制：每增加一个功能，都需要考虑这个功能执行的结果如何用 API 表达，并且得支持将界面操作翻译成 API 语句。

……

也有一些功能看似比较全局，但实际上很容易做正交分解，比如服务端的所有 API 都需要鉴权，都需要记录日志。它们似乎有全局性的影响，但一方面，通常可以在 API 入口统一处理，另一方面就算只是提供辅助函数，具体的鉴权和记录日志都由每个 API 自行处理，心智负担不算太高。所以对于这类功能，我们可以不把它归为全局性功能。

正因为需求交织在一起，全局性功能往往难以彻底进行正交分解。但对于架构师来说，难不代表应该轻易就放弃对正交分解的追求。

不能放过任何一块难啃的骨头。

读盘 / 存盘功能

不能很好分解往往还是需求的分析没有到位所致。前面在 “60 | 架构分解：边界，不断重新审视边界” 这一讲中我们已经拿 “读盘 / 存盘” 作为案例进行过分析。最终我们选择了引入 IO DOM 来进行正交分解。这里面的关键点在于：

其一，“读盘 / 存盘” 本身需求是发散的，因为要支持的文档格式只会越来越多。所以我们必须把它独立成一个子系统，比如叫它 IO 子系统。

其二，既然要独立子系统，就需要抽象出它对核心系统的稳定依赖。为什么这个稳定依赖最后设计为 IO DOM，是因为 DOM 是核心系统的常规界面。

引入只与数据有关的 IO DOM，相当于给 DOM 规范了一个接口子集，用于和 IO 子系统交互。这样的好处是，虽然 IO DOM 是 IO 子系统对核心系统的侵入，但这是没办法的，因为读盘存盘是全局功能，我们没法消除这种全局性，但是可以尽可能削弱到最低。

如果 IO DOM 的确是 DOM 的子集，我们相当于已经找到了尽可能削弱的方法，因为 IO DOM 名称上虽然带了 IO，但是它只是一个归类，实际上这些接口都是核心系统的常规接口，并非为 IO 子系统定制。这样一来，读盘与存盘带来的全局性影响就近乎被消除了。

这是一种很好的思考方式。

全局性功能往往容易带来某种复杂的框架。这不难理解，毕竟它是全局性的，所以常规的思路是为这个功能实现一个库，并建立一套使用它的机制，也就是框架，以应用到核心系统中去。上面 IO DOM 则是反其道而行之，通过抽象核心系统的接口，让全局性功能反向依赖这些接口来完成。这不容易，但是这样做核心系统受到的伤害值最低。

Undo/Redo 功能

我们再看一个例子，比如 Undo/Redo 功能。

读过设计模式的小伙伴们可能都知道，在设计模式中有一个模式叫 Command 模式，专门用于解决 Undo/Redo 这个功能场景的。它的基本思路是，每个用户操作都实现为一个 Command，每个 Command 需要实现反操作，以便做到 Undo 的能力。

这是一个典型的 Undo/Redo 框架。实际上这个框架本身做的事情并不多，基本上就是维护一个 Command 队列，并基于这个队列提供 Undo 和 Redo 功能。

看起来不错的样子，但实际上框架只节省了 1% 的工作量。其余 99% 的工作量在实现一个个 Command 身上，框架使用方的心智负担不是一点点的大。

那么有可能让 Undo/Redo 与核心系统解耦么？

这当然是可能的。

我第一次对 Undo/Redo 实现机制反思的灵感，来自于做 IO 子系统的经历。前面某一讲中我也提过，在我实习的时候，做的第一份工作任务是读盘与存盘。在做需求分析的时候，我发现微软 Office 支持一个很有意思的功能，叫快速存盘。在编辑一份 Word 文档，打几个字存盘时，Word 很快就可以保存完毕，而 WPS 当时则会导致交互界面停顿，存盘没有完成时用户无法编辑。

微软怎么做到的呢？它背后的机制就是快速存盘。所谓快速存盘，就是存盘的时候并不是把完整的文档写到磁盘文件中，而是将上一次存盘到这一次存盘的增量部分，追加到文档的尾部。这样一个 Word 文件就有多个版本的文档，每次读盘的时候只需要读出一个最新版本即可。

当然要想避免系统无法响应用户编辑的另一个思路是异步存盘。也就是在存盘命令执行之初，我对整个文档的 DOM 建立一份镜像（Snapshot），存盘的时候基于镜像进行后台存盘，就不会影响到用户交互。

虽然镜像的实现代价不低，但这个思路有它的独特好处，比如支持异步打印。打印机是比磁盘更慢的 IO 设备。如果在打印的时候用户就没法编辑，也是不太好的用户体验。而打印显然也无法通过类似快速存盘这种机制来实现加速，但镜像功能则可以很好地提升打印的体验。

这些对 IO 子系统的思考，为什么会对我思考 Undo/Redo 机制设计有帮助？因为它们有一个共同点，就是都和数据本身密切相关。

比如 Word 文档支持存储多个版本，我们很容易就想到，其实这个机制可以用来做 Undo/Redo。想象一下，如果用户每进行一次编辑，我就自动执行一次快速存盘，这样就在磁盘中形成多个版本。这样在做 Undo 的时候，我们只需要回退到上一个版本的文档即可。

事实上，只要支持了多版本，就有了镜像能力，也有了 Undo/Redo 能力。

这些思考，就促进了后来数据层（DataLayer）的诞生。怎么理解这个数据层？你可以把它类比为服务端的数据库。它是一个存储中间件，负责托管所有的数据。

中肯地说，数据层的引入有好有坏。

好处不必多言，有了数据层，所有异步操作不是问题，Undo/Redo 不是问题，也还有更多想象空间。

不好的地方是，它是 Model 层的基础，对我们实现 Model 层的业务逻辑是有侵入的。基于内存数据结构写程序，和基于数据库写程序，体验上会有很大的差异。从避免绑定的角度，我们会尽可能将这种差异隐藏起来，把基于数据层与不基于数据层的差异消除。

当然，随着今天软件服务化（SaaS）大行其道，基于某种存储中间件来写业务逻辑，越来越多人意识到它已经是一种必然的趋势。

回顾我们解决 Undo/Redo 的思路，你会发现，它并不是在问题发生的地方解决。这也是需求分析的复杂性所在。

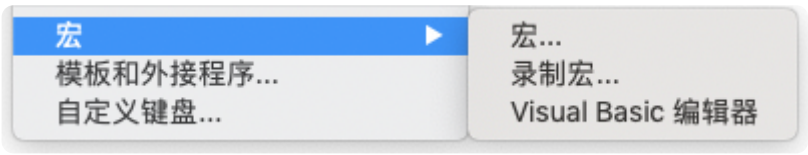
宏录制功能

我们再看 “宏录制” 功能。这个功能使用的人应该不太多，不少人甚至可能并不知道它的存在。要理解 “宏录制”，首先需要知道什么是宏（Macro）。

简单来说，所谓宏（Macro），是指二次开发的代码。微软几乎所有的产品都有二次开发接口，也就是 API 层，典型代表是 Office 和 Visual Studio。

有了二次开发接口，就可以有生态，有围绕着 Office 和 Visual Studio 的生态厂商，来争强产品的能力，也可以让 Office 和 Visual Studio 更容易地融入到企业的业务流中。可以说，支持宏是微软做得最牛的地方。

那么什么是 “宏录制”？简单说，就是把用户的界面操作用 API 调用的方式记录下来，把它变成一段二次开发代码。



这有几点好处。

其一，被录制下来的 “宏”，可以被反复重放，如果某件事情经常发生，它就可以改善我们的工作效率。你甚至可以为 “宏” 指定一个快捷键，这相当于你作为用户，竟然可以给系统添加新功能。

其二，被录制下来的 “宏”，可以进行修改迭代，进行功能的增强。这有助于二次开发的新手学习 Office 或 Visual Studio 的 API 接口，大幅降低二次开发的入门难度。

那么怎么支持 “宏录制”？这个功能和它比较像的是服务端的日志，只是略有不同。

比较像的地方是，宏录制也像日志一样，会去记录一段文本。我们想象一下，如果我们的 Model 层 DOM API 也基于 RESTful API 接口，那么我们就可以在 API 入口的地方去实现

“宏录制”。

不同的地方是，“宏录制” 需要考虑 API 嵌套，我们实现某个 API 可能会调用另外某个 API，但是录制的时候，肯定只能录最外层的 API，而不是所有 API 调用都被录制下来。

这些都比较好解决。所以 “宏录制” 相比前面的 “存盘 / 读盘”、“Undo/Redo” 而言，是一个侵略性相对小的功能，心智负担比较低。

架构师的信仰

通过这些例子，我们需要坚定的一个信念是，任何功能都是可以正交分解的，即使我目前还没有找到方法，那也是因为我还没有透彻理解需求。

这是架构师的信仰。

换句话说，怎么做业务分解？业务分解就是最小化的核心系统，加上多个正交分解的周边系统。核心系统一定要最小化，要稳定。坚持不要往核心系统中增加新功能，这样你的业务架构就不可能有臭味。

这是我们的信仰。重要的话要说三遍。

在模块演化的过程中，随着功能的增加，复杂模块的演化可能会经历剧烈的调整期。通常这种剧烈调整起源于需求理解的进一步深化，引发对原模块接口的反思。无论如何，记住最重要的一点：保持核心系统的纯洁性比什么都重要。

结语

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，也就是我们今天说的全局性功能。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。

但无论如何，我们需要坚持作为一名架构师的信仰：

> 任何功能都是可以正交分解的，即使我目前还没有找到方法。

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大家很熟悉的 “开闭原则（Open Closed Principle，OCP）” 。

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。

© 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　加餐 | 实战： “画图程序” 的整体架构

下一篇　62 | 重新认识开闭原则 (OCP)

精选留言 (11)

写留言

leslie
2019-12-03

"任何功能都是可以正交分解的，如果没有，那是因为没有透彻理解需求"。数据库多年，上次和池老师见面时-他描述他的人生轨迹时，自己得到了一个反思-全栈去看待梳理，然后换位思考去沟通。

大概万事万物离不开数据：数据部门有时反而成了中间点。站在中间点的角度和产品、销售、开发、运维之间做了不少需求的协调和梳理，在不断的换位思考中能理解可能的...

展开

4

Aaron Cheung
2019-12-03

任何功能都是可以正交分解的，即使我目前还没有找到方法，那也是因为我还没有透彻理解需求。

醍醐灌顶

作者回复: 👍

4

清歌
2019-12-04

核心功能最小话，保持稳定；外围功能正交分解，这个总结太厉害了

1

tt
2019-12-08

“业务分解就是最小化的核心系统，加上多个正交分解的周边系统。核心系统一定要最小化，要稳定。坚持不要往核心系统中增加新功能，这样你的业务架构就不可能有臭味。”

我觉得这句话就是下节课的引子，核心是最小化的，就可以容易做到对修改封闭，因为它是业务的本质，除非业务变了，或者是我们没有彻底理解业务，否则它不会剧烈变化。...
展开

作者回复: 说得很赞👍

CoderLim
2019-12-06

老师一直在强调业务正交分解，不理解，能否说一下如何做

作者回复: 这块和具体业务相关，关键做好需求分析。

诗泽
2019-12-05

当一个新需求来了或者要开始一个新项目的时候架构师在做需求分析和架构设计的时候其他工程师们在做什么?

展开

作者回复: 在做老项目

Fs
2019-12-04

有点抽象，需要更多慢慢咀嚼消化

展开

阿火
2019-12-04

正在反复阅读当中，正交分解

展开

K战神
2019-12-04

大道至简，
把事情做明白想明白还得千锤百炼，
还得深刻思考中千锤百炼地做明白想明白，
回头再看，大道至简。
...

展开

睡觉🐷
2019-12-03

任何功能都是可以正交分解的，即使我目前还没有找到方法。

梦醒十分
2019-12-03

干货满满呀！

展开