66 | 架构老化与重构

2019-12-20 许式伟

讲述：姚迪迈

时长 12:36 大小 11.56M

你好，我是七牛云许式伟。

在 “64 | 不断完善的架构范式” 这一讲中，我们强调了架构师在日常工作过程中不断积累和完善架构范式的重要性。而上一讲 “65 | 架构范式：文本处理” 则以我个人经历为例，介绍了文本处理领域的通用架构范式。

架构的老化

架构的功夫全在平常。

无论是在我们架构范式的不断完善上，还是应对架构老化的经验积累上，都是在日常工作过程中见功夫。我们不能指望有一天架构水平会突飞猛进。架构能力提升全靠平常一点一滴地不断反思与打磨得来。

今天我们要聊的话题是架构老化与重构。

架构老化源于什么？

在我们不断给系统添加各种新功能的时候，往往会遇到功能需求的实现方式不在当初框架设定的范围之内，于是很多功能代码逸出框架的范围之外。

这些散落在各处的代码，把系统绞得支离破碎。久而久之，代码就出现老化，散发出臭味。

代码老化的标志，是添加功能越来越难，迭代效率降低，问题却是持续不断，解决了一个问题却又由此生出好几个新问题。

在理想的情况下，如果我们坚持以 “最小化的核心系统 + 多个相互正交的周边系统” 这个指导思想来构建应用，那么代码就很难出现老化。

当然，这毕竟是理想情况。现实情况下，有很多原因会导致架构老化难以避免，比如：

软件工程师的技术能力不行，以功能完成为先，不考虑项目的长期维护成本；

公司缺乏架构评审环节，系统的代码质量缺乏持续有效的关注；

需求理解不深刻，最初架构设计无法满足迭代发展的需要；

架构迭代不及时，大量因为赶时间而诞生的补丁式代码；

……

那么，怎么应对架构老化？

这个问题可以从两个视角来看：

该怎么重构系统，才能让我们的软件重新恢复活力？

在重构系统之前，我们应该如何进行局部改善，如果增加新功能又应该如何考虑？

我们先聊后者，毕竟重构系统听起来是一件系统性的工程。而添加新功能与局部调整则在日常经常发生。

老系统怎么添加新功能

先说说添加新功能。

正常来说，我们添加功能的时候，尤其是自己加入项目组比较晚，已经有大量的历史代码沉淀在那里的时候，通常我们应该把自己要添加的功能定位为周边功能。对于周边功能，往往考虑最多的点是如何少给核心系统添加麻烦，能够少改就少改。

但是，这其实还不够。实际上当我们视角放在周边系统的时候，其实它本身也应该被看作独立业务系统。这样看的时候，我们自然而然会有新的要求：如何让新功能的代码与既有系统解耦，能够不依赖尽量不依赖。

这个不依赖是有讲究的。

不依赖核心的含义是业务不依赖。新功能的绝大部分代码独立于既有业务系统，只有少量桥接的代码是耦合的。

实际上对于任何被正交分解的周边系统 B 与核心系统 A，理想情况我们最终得到的应该是三个模块：A、B（与 A 无关部分）、A 与 B 桥接代码（与 A 相关的部分）。虽然从归属来说，A 与 B 桥接代码我们通常也会放到 B 模块，但是它应该尽可能小，且尽可能独立于与核心系统无关的代码。

理解这一点至关重要。只有这样我们才能保护自己的投资，今天开发新功能的投入产出可以最大程度得以保留。未来，万一需要做重构，我们的重构成本也能够尽可能最小化。

不依赖的另一个重要话题是要不要依赖公司内部的基础库。这一点需要辩证来看，不能简单回答依赖或不依赖。完全不依赖意味着放弃生产力。

这里基本的判断标准是，成熟度越高的基础库越值得依赖。成熟度的评估依赖于个人经验，首先应该评估的是模块规格的成熟度，因为实现上的问题让时间来解决就行。模块规格是否符合你的预期，以及经过了多少用户使用的打磨，这些是评估成熟度的依据。

还是以我做办公软件时期的经历为例。从重构角度来说它很典型，既有的代码有几百万行。我第一个做读盘与存盘之外的新功能是电子表格的智能填充。这个功能比较常用，用户可以选择一个区域，然后移动鼠标到被选区域右下角，在鼠标变成十字时，按下鼠标左键不放并移动鼠标以进行单元格内容的自动填充。填充方向是上下左右都可以。

我怎么做这个功能？首先是实现一个基本纯算法的模块，输入一个值矩阵（可以是数值、日期，也可以是字符串等），要预测的序列个数，输出对应预测的值矩阵。为什么自动填充的方向在算法这里消失了？因为我们按填充方向构建值矩阵，而不是用户屏幕上直观看到的矩阵。

然后抽象了核心系统的两个接口，一个是取一个区域的单元格数据，包括值和格式，一个是设置一个单元格的值和格式。基于这个抽象接口，我们实现了完整的自动填充逻辑。

最后，是对接这个自动填充模块与既有的业务系统。从 Model 层来说，只需要在既有的业务系统包装对应要求的接口即可。而且取区域单元格、设置一个单元格的值，这些是非常通用的接口，无论既有系统长什么样，我们都可以轻松去实现所需接口。

这就是做新功能的思路，尽可能与既有系统剥离，从独立业务视角去实现业务，抽象对环境的依赖。最后，用最少量的对接代码把整个系统串起来。

架构的局部优化

聊完添加新功能，我们谈谈局部调整。它的目标是优化某个功能与核心系统的耦合关系。

局部调整看似收效甚微，但是它的好处是可以快速推动。而且，日拱一卒，如果我们能够坚持下来，最后的效果远比你想象得好。

它有两种常见做法。

一种是重写，或者叫局部重构。它相当于从系统中彻底移除掉与该功能相关的代码，重新写一份新的。这和开发一个新功能没什么两样，最多看看被移除的代码里面，有哪些函数设计比较合理，可以直接拿过来用，或者稍微重新包装一下能够让规格更合理的。

但是我们不能太热衷于做局部重构。局部重构一定要发生在你对这块代码的业务比较了解的情形，比如你已经维护过它一阵子了。

另外，局部重构一定要把老代码清理干净，不要残留一些不必要的代码在系统里面。剩下来的事情，完全可以参考我上面提的实现新功能的方法论来执行。

另一种是依赖优化。它关注的重心不是某项功能本身的实现，而是它与系统之间的关系。

依赖优化整体上做的是代码的搬运工。怎么搬代码？和删除代码类似，我们要找到和该功能相关的所有代码。但是我们做的不是删除，而是将散落在系统中的代码集中起来。我们把对系统的每处修改变成一个函数，比如叫 doXXX_yyyy。这里 XXX 是功能代号，yyyy 则依据这段搬走的代码语义命个名。

你可能觉得这个名字太丑了。但是某种程度来说这是故意的。它可以作为团队的约定俗成，代表此处待重新考虑边界。

不要理解错了，它不是说我们需要重新思考我们现在正在做代码优化的功能边界。它是说我们要重新考虑核心系统的边界。尤其是如果某个地方有好几个功能都加了 doXXX_yyyy 这样的调用，这就意味着这里需要提供一个事件机制，以便这些功能能够进行监听。而一旦我们做了这件事，你就发现核心系统变得更稳定了，不再需要因为添加功能而修改代码。而这不正是 “开闭原则（OCP）” 所追求的么？

回到我们要进行依赖优化的功能。集中了这个功能所有代码后，这个功能与系统的耦合也就清楚了。有多少个 doXXX_yyyy，就有多少对系统的伤害（参阅 “58 | 如何判断架构设计的优劣？” 中的伤害值计算）。

如果伤害值不大，代表耦合在合理范围，做到这一步暂时不再往下走是可接受的。如果耦合过多，那就意味着我们需要站在这个功能本身的业务视角看依赖的合理性了。如果不合理，可以考虑推动局部重构。

所以，局部重构不应该很盲目，而应依赖于基于 “伤害值” 的客观判断。习惯于在不理解的情况下就重构，这实在不太好。认同他人是很重要的能力修炼。况且作为架构师，事情优先级的排列是第一位的，有太多重要的事情值得去做。

依赖优化的好处比较明显。其一，工作量小，做的是代码搬运，不改变任何业务逻辑。其二，可以不必深入功能的细节，只需要找到该功能的所有相关代码，这是难点，然后把它们集中起来。

尽可能把我们认为非核心系统的功能，都基于依赖优化的方式独立出去。这样核心系统与周边系统的耦合就理清楚了。

依赖优化，可以把周边系统对核心系统的代码注入，整理得清清楚楚。这是事件机制的需求来源。

依赖优化也能够及时发现糟糕的模块，和核心系统藕断丝连，斩不断理还乱，这时我们就需要对这个功能进行局部重构。

核心系统的重构

完成这些，我们下一步，就要进入重构的关键阶段，进行核心系统重构。

对于一个积弊已久的系统，要想成功完成整体的重构是非常艰难的。

如果我们一上来就去重构核心系统，风险太高。一方面，牵一发而动全身，我们无法保证工程的交付周期。另一方面，没有谁对全局有足够的了解，重构会过于盲目，项目的执行风险难以把控。

确定要对核心系统进行重构，那么最高优先级是确定它的边界，也就是使用界面（接口）。

能够在不修改实现的情况下调整核心系统的使用界面到我们期望的样子是最好的。

周边系统对核心系统的依赖无非两类：一是核心系统的功能，表现为它提供的 DOM 接口；二是核心系统提供的事件，让周边系统能够介入它的业务流程。

对所有周边模块进行依赖优化的整理，细加分析后可以初步确定核心系统需要暴露的事件集合。

进一步要做的事情是把核心系统的 DOM 接口也抽象出来。这一步比较复杂。它包含两件事情：

- 让周边系统对它的依赖，变成依赖接口，而非依赖实现；
- 审视核心系统功能的 DOM 接口的合理性，明确出我们期望的接口设计。

我们可以分步骤做。可以先做实现依赖到接口依赖的转变。这有点像前面依赖优化的工作。只不过它不是搬代码，而是把周边模块独立出去，将它与核心系统的依赖关系全部调整为接口。这样，不管抽离出来的 DOM 接口是否合理，至少它代表了当前系统的模块边界。

这一步做完，理论上 mock 一个核心系统出来和周边系统对接也是可行的。只不过可能这个 DOM 模型太大，要 mock 不那么容易。

接下来，就是最重要的时刻。

我们需要对核心系统的接口进行重新设计。这一步的难点在于：

第一，我们对业务的理解的确有了长足的进步。我们抽象的业务接口有了更加精炼符合业务本质的表达方式，而不是换汤不换药，否则我们就需要质疑这次重构的必要性。

第二，对周边系统切换到新接口的成本有充足的预计。对周边系统来说，这是从老接口过度到新接口的过程。虽然理论上让核心系统维护两套 DOM 接口同时存在，在技术上是可行的，但是这个过渡期不能太长，否则容易让人困惑，不清楚我们倡导的是什么。

完成了接口改造，剩下来就简单了。核心系统，每一个周边系统，彼此完全独立，可以单独调整和优化。嫌当前的核心系统太糟糕？那就搞搞。为什么可以这么轻松决策？因为就算我们要重新写核心系统，要做的事情也很收敛，不会影响到大局。

这不像那些系统边界分解得不清不楚的业务系统。要改核心系统的代码？

不要命了么？

结语

重构工作是很有技巧性的，很能培养一个人的架构能力。做多了，我们可以建立对代码耦合的条件反射，看一眼就知道架构是否合理。

但重构不是技巧性那么简单。

实际上从难度来说，重构比一个全新业务的架构过程要难得多。重构，不只是一个架构的合理性问题。它包含了架构合理性的考量，因为我们需要知道未来在哪里，我们迭代方向在哪

里。

但重构的挑战远不只是这些。这是一个集架构设计（未来架构应该是什么样的）、资源规划与调度（与新功能开发的优先级怎么排）、阶段规划（如何把大任务变小，降低内部的抵触情绪和项目风险）以及持久战的韧性与毅力的庞大工程。

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 “架构思维篇：回顾与总结” 。

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　65 | 架构范式：文本处理

精选留言 (6)

写留言

丁丁历险记
2019-12-20

正在重构一个运营七年的盈利项目，举步维艰

展开

作者回复: 一起加油

4

leslie
2019-12-21

引用老师课程中关于重构的一句经典话语"架构设计（未来架构应该是什么样的）、资源规划与调度（与新功能开发的优先级怎么排）、阶段规划（如何把大任务变小，降低内部的抵触情绪和项目风险）以及持久战的韧性与毅力的庞大工程。”体现了老师一直强调的架构与业务的理解。

软件架构的老化与重构参与不多：不过数据库架构这块的事情经历过不少。虽范围...

展开

作者回复: 挺好的思考与总结

Aaron Cheung
2019-12-20

许老师的讲解让我明白 talk is really important

展开

梦醒十分
2019-12-20

要多看几遍呀！

展开

沫沫（美丽人生）
2019-12-20

许老师，需求的变化点和稳定点，可以作为判断系统的核心模块和周边模块的依据吗？

作者回复: 有这个味道在里面

1

沫沫（美丽人生）
2019-12-20

许老师，想请教一个核心功能界定的问题，还有一个产品之间差异化的问题。怎么确定一个系统的核心模块呢？作为竞争对手，WPS和office的差异化是什么？怎么在程序架构中抽象这种差异化呢？望您不吝赐教

作者回复: 需求越稳定的部分，越处于核心的位置。MVC 框架，MV 是核心，C 是周边。M 如果比较大，内部还可以分核心与周边。wps 和 office 的竞争很有意思，从最初的不一样的 office 到后面提一样的 office，你可以感受一下。程序架构不会受此类商业策略的影响。

1