37｜能力维度二：如何提升解决横向问题的能力？

2022-05-17 郭东白

《郭东白的架构课》 课程介绍 >

讲述：郭东白

时长 17:57 大小 16.44M

你好，我是郭东白。我们上节课讲了，程序员的结构化设计能力是向架构师过渡的重要基础。假设你现在已经拥有了这项基础能力，想开启自己的架构师职业生涯新篇章，那么该从开始呢？

这节课我们就来讨论一下这个话题。

从程序员到兼职架构师

我们先研究一下程序员和兼职架构师这两个角色的区别：

> 在软件架构这个上下文里，程序员指的是能够结构化地完成业务需求的一线软件工程师。

从定义的描述中，我们很轻松就能看到两个角色的差异主要是在**横向问题**上。也就是说，兼职架构师这个角色，除了需要关注自己的代码外，还要关注横向问题。

横向问题，简单来说就是**软件系统内部与业务无关的技术债**，比如性能、可扩展性、可用性、可测试性、可维护性和安全合规等问题。这些问题都属于非功能性需求，也就是说，产品经理一般不会把这些问题直接写在需求文档里。

技术债的产生一般有两个原因：

1. 你与周围同事在横向问题领域内有认知盲区；
2. 由于日常研发过程中的优先级取舍，导致一些技术债被短期搁置。

可是日积月累，这些技术债必然会成为整个团队的负担，影响软件的整体质量。这个时候你就要意识到：机会来了！

那么怎么把握住这个机会，解决这些技术债呢？我们在模块导读中提到了，从程序员成长为兼职架构师，需要跨越的主要的能力障碍就是**责任边界**。

和上节课提到的习惯性思考一样，责任边界也属于意识和习惯上的问题。解决横向问题并不是你这个研发人员的责任，毕竟很少有公司会为此配备专属的研发资源。多数人都没必要突破自己的责任边界，去解决公司的横向问题。所以你需要找到一个能对日常研发任务形成促进作用的支点问题，帮助自己突破这个障碍。

不过，解决横向问题需要具备相应的专业知识。如果你处在职业生涯初期，每天被业务需求压得喘不过气，哪里有时间去学习研究横向问题呢？

而且，即使你愿意牺牲个人休息或娱乐时间，**做选择**也并不是一件容易的事情。横向问题有很多，包括性能、可测性、可扩展性、安全等。那么从哪一个问题开始，会是更好的选择呢？还是每样都学一点儿呢？

横向能力呢？

类似这样的选择，在架构师的职业发展中比比皆是，都需要我们作出良好的取舍。

如何跨越从程序员到兼职架构师的障碍？

我们先讨论这个最紧迫的问题。

答案是**不应该**。想想看，如果连日常的工作做不好，甚至还可能在下一批淘汰的候选名单里。在这种情况下，上级不可能把任何有挑战的横向问题交给你。所以你的首要任务，一定是先把日常的研发工作做好。

如果有额外的精力，我应该先花费精力提升本职工作的深度，还是分出一部分精力去提升解决横向问题的能力呢？

这是个好问题。就像我在开篇词中提到的，如果你的战略意图是做一个优秀的架构师，那么这个问题的答案是肯定的。当然，如果你志不在此，那么解决横向问题就不是最好的选项。对于这个问题，我们在模块一最大化商业价值里有一些探讨，你可以学习参考。

现在，假设你下定决心要在横向问题上投入精力了，那么接下来要面临的问题就是：学**习解决横向问题，该从哪里开始呢？**

在这个决策的过程中，我一般会从三个方面来综合考虑。

首先是**个人兴趣。**在刚开始时，我认为个人兴趣最重要。因为我们肯定要花费个人的休息娱乐时间来学习这个领域的专业知识，提升自身能力的稀缺性。如果没有足够的兴趣，这个过程肯定是不那么愉快的，也很难坚持下来。

其次，**要考虑商业价值。**也就是说，解决这个横向问题能给公司带来多大的价值？给公司带来的价值越大，未来你获得的机会回报也就越大。甚至，公司还愿意花钱来加速你的能力提升。

下，我建议不要在这个领域投入。没有高质量的实践，你的学习很难有用武之地。

无论选择什么方向，都要看清楚自己的相对优势所在。我再强调一下，是**相对于团队其他同学的优势**。作为一个兼职架构师，最重要的价值就是帮助团队中被某个横向难题挡住的程序员清理路障。你的相对价值越大，获得的实践机会就越多，横向知识的雪球就会越滚越大。

不过，如果现在还没有特别感兴趣的方向，公司内部的机会也差不多，只是单纯想找一个领域先提升自己。在这种情况下，我建议**从稳定性开始**。

为什么要从稳定性开始？

首先，稳定性是任何一个企业都绕不开的问题，**是第一性的**。企业既然花钱投入了软件研发，最终肯定要做一个稳定的软件。

其次，做稳定性会先**让自身受益**。请想象一下，如果你写的代码和服务更稳定可靠，就不用半夜从床上爬起来去接报警，活得也更轻松一些。这样才有机会脱离工作的高压，去研究一些更有深度的问题。

最后，稳定性问题和程序员日常代码工作的连续性比较大，你可以一边提交日常需求，一边做稳定性相关的改造。这样一来，获取一些实践机会，几乎不需要任何额外的授权，同时也能从实践中不断获得信心。

稳定性这个话题非常大，将来有时间我打算专门开个专栏来讲。在今天这节课，我先把之前总结的一些经验分享出来。当时是用英文写的，这里我加上一些简单的中文注释。

1. **Trust none !**

> 永远不要轻信（别人的话）！意思是，你依赖的服务，不会返回他们应该返回的东西写代码检查依赖，远比 Debug 代码容易得多。 这条法则同样适用于安全问题。

2. **Only rely on the most reliable.**

3. **Everything decays, especially data.**

> 所有东西都会过期变质，尤其是数据。意思是，你拿到的数据、配置、服务、消息通知等都可能过期。你给别人的数据也是一样的。这些数据可能在过去某个时间点是有效的，但绝对不是现在。

4. **When it comes to survival, everything can be downgraded.**

> 在生存（也就是维持系统可用性）面前，所有依赖都可以被降级。

5. **Availability without cost consideration is bullshit.**

> 不考虑成本的稳定性就是耍流氓。

6. **Save yourself!**

> 猪队友永远在你最危急的关头出现，关键时刻要有能力保护自己！

你可能会问，这六条经验原则好用吗？

从我个人的经验来看，的确是好用的。这是一个相对成熟的互联网企业员工的稳定性生存实用指南，每一条法则都是靠大量的真实案例训练出来的。多年前我在亚马逊工作时，阅读了 400 多篇 P0 故障复盘（COE），发现其中 80% 以上的故障，靠这几条法则就能完全避免或者快速恢复。

我来剖析一下这六条经验原则的大致出处和背景，以及它们曾经给我带来的价值。

第一条原则是 Trust none，主要源于我在甲骨文工作期间，公司要求人人都要学习的代码安全规范。不过这里也包含 Defensive Coding 的核心理念。

这条原则基于一个非常重要的认知：代码可靠的原因在于，**从来都不要相信自己的代码是在一个可靠的环境中运行**。也就是说，任何时候都要坚持检测你的依赖方，确保他们的

环境问题了。

第二条原则是 **Only rely on the most reliable**，来自系统工程学科中的一个结论，也就是系统的可用性会随着复杂度的提升而降低。如果想设计一个高可用系统，就必须把依赖最小化。

除此之外，从这句话我们又可以引申出两个结论。第一，如果是被迫引入依赖的话，就要选择最靠谱的人和最靠谱的模块。这个结论在互联网行业尤其适用。互联网行业的请求、服务质量、程序员的能力分布，大都符合 Zipf Law 原则，也就是 2-8 原则背后的数学规律。所以一定要选择依赖最靠谱的那一部分。

第二，从管理者的角度来看，在故障出现之前就要锁定那些能真正保护好系统，并且能给出正确判断和方向的**人**。在稳定性治理这个领域，当所有“聪明”的方法（暗指不借用人力的方法）都用尽之后，还有最后一根救命稻草——人力恢复系统的可用性。所以要不断招聘、培养、训练和保护好具备这种能力的人。可以说，这句话是对稳定性响应原则的总结。

第三条原则，是从我在亚马逊的工作经历中总结出来的。在分布式的世界里，那些简单回滚解决不了的生产环境问题，往往是因为数据配置和频繁更新的代码不匹配而造成的。对于这部分问题，需要把线上系统和可能已经被污染的数据尽量隔开，快速止血。这其实是个设计原则，也就是在设计中要考虑数据污染的情形。如果有多个选择的情况下，应该选择范围更小、可靠性更高的数据来源。

第四条原则，是对亚马逊 D-Day 和阿里双 11 高可用性保障方法论的总结。也就是通过大量的压测和充足的预案，确保一个超高赌注事件能 100% 成功。这个话题，我们在模块二可行性探索环节中有详细的解释，就不再重复了。

第五条原则，是我这些年对一些公司过度稳定性建设的反思，更多是从中小企业 CTO 的视角来重新审视稳定性的投入。

在中小企业，做稳定性必须要控制成本，而不能因为追求稳定性而牺牲迭代的速度或者大量的现金。在稳定性建设中，要尽量避免完美。世界上不存在 100% 高可用的系统，任何

最后一条原则是故障响应的保底原则，意思是先自救。就像乘坐飞机时要求先准备好自己的氧气面罩，然后才能去帮助他人一样。当故障发生的时候，首先要想方设法地恢复自己所负责的服务，不要等待强依赖方的恢复。否则你会沮丧地发现，自己竟然成了木桶的最短板。

你可能会问，怎么没有从头到尾的稳定性建设原则啊？这不完备啊？监控报警建设都没提。

降级、限流、监控、报警等稳定性常规手段，在多数互联网公司都已经齐备。甚至在互联网早期不齐备的时候，我也不会把这些手段当成原则。因为这都是手段。**原则的价值就在于告诉我们如何做取舍，丢车保帅，**而不是面面俱到。

我的职业发展历程中，做稳定性就是靠这六条法则来保命的。

前面也提到了，假设你所在的公司里已经有不少稳定性大拿，或者所处的环境中没有较多的稳定性机会，就应该选择其他的横向问题去提升。

除了稳定性之外，我认为**性能**的优先级也比较高。主要是商业回报容易量化，也容易与日常工作形成正向促进。此外，做性能也可以让自己对代码和公司业务更加熟悉。

横向问题其实有很多，我的观察是多数研发人员都不缺少成长的机会，缺少的往往是**意愿**。只要平时不断积累，主动解决问题，机会永远比竞争多。

最后我再强调一点，横向能力不是积攒得越多越好。公司发展到比较大的时候，相比覆盖广度，横向能力的深度和稀缺性要更重要一些。

小结

像生活中的很多事情一样，像生活中的很多事情一样，从程序员过渡到架构师，需要有够的意愿和投入。具体怎么做，我相信随着技术的变化也会发生变化。因此在这个过程中，你的主动规划更重要。

如果你是个利益驱动的人，那么回报就决定了你的投入度。回报是由解决横向问题带来的商业价值决定的，人才供给竞争则是人才市场对回报的自然反应。也就是说，市场会带来人才供给的弹性，高回报的场景必然会有更多的人才涌入，最终回报和投入将趋于理性。不过无论如何，兴趣可以让你以更多的投入来维持自己的稀缺性。后面的课程还会介绍跨越其他障碍的办法，策略雷同，我就不再重复今天的描述了。

我在这节课还分享了自己总结的稳定性法则。正如我们在模块一里面反复强调的，我分享它们的目的不是让你去记忆，而是请你也尝试用同样的方法在工作中不断总结，然后用你的案例去修正它们。我也期望你能把反例分享在留言区，我们一起讨论提升。

思考题

三个作业，你可以选其中一个做一下。还是和上次一样，目标不是做得全，而是做得深，做的有感触。也欢迎你把你的感想分享给大家。

1. 如果你曾经跨越过从程序员到兼职架构师这个障碍，能分享一下最大的心得吗？
2. 请思考一下，你现在团队里面的横向领域中，最大的技术机会在哪里？为什么是这个机会呢？其他人扑上去了吗？
3. 请总结一下最近半年来，你遇到的稳定性重大问题。有哪些稳定性问题，是不能被我提到的六个原则所解决的？如果要增加一个原则，你的建议是什么？如果要删除其中一条原则，你会删除哪一个？为什么？

欢迎把你的思考和想法分享在留言区，我会和你交流。也欢迎把课程分享给你的朋友或同时，我们一起进步。同时也邀请你关注我的抖音号“郭东白”。下节课再见！

上一篇 36 | 能力维度一：如何提升结构化设计的能力？

下一篇 38 | 能力维度二：如何提升解决跨领域冲突的能力？

精选留言

写留言

> 由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。