客户需求重于个人简历

Don't Put Your Resume Ahead of the Requirements

尼廷·博万卡（Nitin Borwankar）

作为工程师，我们常常要向客户推荐技术、手段，甚至方法论来解决问题。但有时我们心里不是想寻求解决问题的最佳方案，而是希望借此丰富自己的简历。这样做很可能得不偿失。

积累一批满意的客户，选择切合实际的技术解决他们的难题，让他们乐于推荐你，才是最好的履历。信誉远胜过时髦的编程技巧和流行的范式。掌握最新的技术趋势，与时俱进固然重要，但不能让客户为此买单。作为架构师，职业操守绝不能忘。公司托付重任给你，是期望你恪尽职守，不受利益诱惑。如果你觉得项目不够尖端，挑战性不足，无法满足职业发展的需要，大可另栖高枝，另谋高就。

万一你别无选择，必须参与这样的项目，不要为简历所累。忍痛割爱放弃时髦光鲜的方案确实不容易（哪怕它们并不适合当前的项目），但只有脚踏实地替客户着想，最后才能皆大欢喜。

选择正确的解决方案可以降低项目的压力，团队工作起来更开心，客户也更满意。你会有更充裕的时间，既可以钻研现有技术，也可以利用空闲时间学习新知识，甚至重拾向往已久的业余爱好。家人察觉你的变化后，也会感到欣慰。

把客户的长远需求摆在自己的短期利益之上，才能立于不败之地。

作者简介：

尼廷·博万卡 20 世纪 90 年代初曾先后就职于 Ingres 公司和 Sybase 公司。他曾参与开发最早的网络数据库应用，那时的开发语言是 SybPerl 和 OraPerl，之后他又投身研究 Enterprise Java。他曾积极参与 New-EDI 项目——一个基于 IETF 标准的因特网电子数据交换系统。1994 年以来，他一直作为独立顾问和研究者关注企业数据集成和消息传递的研究。他目前的兴趣包括研究数据库模式（schemas）用来实现企业应用中的大众分类方法（folksonomy），以及将社交网络运用于企业应用时的底层数据库问题。他是数据可移植性（Data Portability）策略小组的成员，负责起草有关用户数据权利的最终用户许可协议。他撰写了多篇数据库方面的文章，发表在 GigaOm.com 和他的博客（*http://tagschema.com*）上。他还拥有一项跨越可信赖边界传递协同消息的专利。

简化根本复杂性，消除偶发复杂性

Simplify Essential complexity; Diminish Accidental Complexity

尼尔·福特（Neal Ford）

根本复杂性（essential complexity）指的是问题与生俱来的、无法避免的困难。比如，协调全国的空中交通就是一个“天生的”复杂问题，必须实时跟踪每架飞机的位置（包括飞行高度）、航速、航向和目的地，才能预防空中和地面上的冲突。像天气骤变这样的情况会令航班计划全盘失效，航班时刻表必须适应不断变化的环境才能避免乘客滞留。

与之相反，偶发复杂性（accidental complexity）是人们解决根本复杂性的过程中衍生的。目前陈旧的空中交管系统，就是一个偶发复杂性的例子。系统设计的初衷是管理数以千计的飞机参与的交通活动，即解决根本复杂性，但是解决方案本身带来了新的问题。事实上，目前正在服役的空中交管系统，其复杂臃肿已经到了难以改善的地步。在全球多数地区，空中交管系统仍然在使用三十多年前的技术。

许多软件框架和厂商提供的“解决方案”都表现出偶发复杂性的症状。解决特定问题的框架很管用，但设计过度的框架增加的复杂性反而超过了它应该缓解的复杂性。

开发人员痴迷于复杂的问题，好比飞蛾喜欢扑火。谁能拒绝迅速解决复杂问题带来的快感？但是开发人员应该解决问题，而不是解谜取乐。在大型软件项目中，关注根本复杂性，消除偶发复杂性，抽丝剥茧制订解决方案，才是真正的挑战。

该怎么做呢？应该尽量选择源自实际项目的框架，警惕那些象牙塔里的产品；分析方案中有多少代码直接用来解决业务问题，有多少只是用来实现用户与应用之间的交互；谨慎使用软件厂商在幕后推动的方案，它们并非一无是处，但往往包含偶发复杂性；要量体裁衣，为问题制订“合身”的解决方案。

架构师的责任在于解决问题的根本复杂性，同时避免引入偶发复杂性。

作者简介：

尼尔·福特是 ThoughtWorks 公司的软件架构师和文化基因探索者（meme wrangler）。ThoughtWorks 是一家全球性的 IT 咨询公司，专注于端到端的软件开发与交付。除了开发软件，尼尔还编写教材，撰写杂志文章，制作课件和视频 DVD，并出版了五本著作。他经常参加各种会议并发表演讲。更多详情，请访问他的个人网站（*http://www.nealford.com*）。

关键问题可能不是出在技术上

Chances Are, Your Biggest Problem Isn't Technical

马克·兰姆（Mark Ramm）

简单的项目（比如工资管理系统）也会翻船，而且这不是个别情况。

为什么？难道是因为我们用错了技术吗？因为错选了 Ruby 而不是 Java，错选了 Python 而不是 Smalltalk？或者选择了 Postgres 而不是 Oracle？还是本该用 Linux 时，错选了 Windows？一旦项目失败，技术往往沦为替罪羊。但是有多少问题真的是 Java 无法胜任的呢，这种可能性有多大？

大多数项目是由人完成的，人才是项目成败与否的基础。如何帮助团队成员完成项目，这个问题很值得静下心来好好思考。

如果团队里有人工作方式不正确，拖项目的后腿怎么办？有一种非常古老但很完善的技术可以帮助你解决问题。它可能是人类历史上最重要的技术创新，这就是对话。

仅仅了解对话的用途还不够。学会尊重他人，给予团队成员充分的信任，是聪明的架构师获得成功必须掌握的核心技能。

关于对话的技巧非常多，但有几个简单的技巧可以显著改善对话的效果：

- 不要把对话当成对抗。

  如果你能看到他人的优点，并把沟通视为请教问题的机会，就会有所收获，同时也能避免引起对方的戒备之心。

- 不要带着情绪与人沟通。

  当你处于愤怒、沮丧、烦恼，或者慌张的情绪中时，对方很可能会误认为你的举动不怀好意。

- 尝试通过沟通设定共同的目标。

  有些人开会时喋喋不休影响别人发言，与其命令他闭嘴，不如请他协助你提高其他人的参与度。告诉他有些同事比较内向，发言前需要安静地理清思路。请他在每次发言之前稍做等待，让同事有机会表达意见。

首先与同事达成一致的目标，把处理冲突和矛盾的过程视为学习的机会，控制住自己的情绪，那么每次沟通都会有所收获，你会做得越来越好。

作者简介：

马克·兰姆是 TurboGears 2 开源项目领头人。这位老兄热爱 Python，喜欢挑战，他干过各种奇怪的工作，除了软件架构师和网络管理员，他还捕过龙虾，在飞车党酒吧当过清洁工。他目前致力于开发软件工具，帮助专业程序员和业余程序员更好地开展工作。

以沟通为中心，坚持简明清晰的表达方式和开明的领导风格

Communication Is King; Clarity and Leadership, Its Humble Servants

马克·理查兹（Mark Richards）

软件架构师普遍喜欢坐在象牙塔里，命令开发人员执行他的命令和技术决策。这很容易引发大家的抵触情绪，造成团队不和，甚至导致产品与最初的需求相去甚远。软件架构师应该想办法提高自己的沟通技巧，帮助大家理解项目的目标。关键在于明确有效的沟通和开明的领导风格。

沟通必须简明清晰。没有人愿意阅读冗长的架构决策文档，架构师言简意赅地表达观点是项目成功的必要条件。项目启动之初，凡事能简则简，千万不要一头扎入冗长的 Word 文档里。可以借助工具，比如简单的 Visio 图表来表达你的想法，尽量画简单些，毕竟时过境迁，想法总会变化。非正式的白板会议是另一种有效的沟通手段，把开发人员（还有其他架构师）召集起来，在白板上写下你的想法，比任何方法都来得有效。此外，别忘了随身携带相机，拍下白板上的内容，通过 Wiki 在团队内共享，毕竟会后回忆讨论的内容并不容易。扔掉冗长的 Word 文档，想办法让大家接受你的观点，最后别忘了详细记录讨论结果。

还有，架构师往往忽略了自己也是领导者。作为领导者，我们必须获得同伴的尊敬才能顺利开展工作。如果开发人员对项目蓝图和决策过程一无所知，必定会产生隐患。安排一位你信得过的开发人员牵头，创造良好的合作环境，请大家共同验证你的架构决策。让开发人员参与架构的制订过程，他们才会买你的账。与其和开发人员对着干，不妨与他们合作。请记住，所有的团队成员（包括质量控制小组、业务分析员、项目经理，以及开发人员）都渴望明确的沟通和开明的领导。只有这样才能改善沟通效果，建立团结健康的工作环境。

以沟通为中心，坚持简明清晰的表达方式和开明的领导风格。

作者简介：

马克·理查兹是 Collaborative Consulting 有限责任公司的主管和高级解决方案架构师，他的主要工作是利用 J2EE 和相关技术为金融服务行业设计并提供大规模面向服务的架构。自从 1984 年进入软件业以来，他在 J2EE 的架构和开发，面向对象设计和开发，以及系统集成方面积累了丰富的经验。

架构决定性能

Application Architecture Determines Application Performance

兰迪·斯塔福德（Randy Stafford）

架构决定应用的性能，似乎是大家都明白的道理，但是事实并非如此。有些架构师认为简单地更换底层软件架构（Software Infrastructure）就足以解决应用的性能问题。他们很可能轻信了“经测试产品性能超出竞争对手 25% ”一类的商业噱头。假设某产品完成特定操作耗时 3 毫秒，竞争对手需要 4 毫秒，这 1 毫秒（25%）的优势如果放到一个性能效率极低的架构里，几乎可以忽略不计。架构是决定应用性能的根本因素。

撇开 IT 经理和厂商的测试团队，另一些人（比如产品技术支持部门和应用性能管理文献的作者）则建议直接通过“调优”（Tuning）架构来解决问题，例如改变内存的分配方法、调整连接池或线程池的大小，等等。但是，如果应用的部署方案满足不了预期的负载（load）要求，或者应用软件的功能架构不能充分利用计算资源，那么无论怎样“调优”都无法带来理想的性能和可伸缩（scalability）特性。这时必须重新设计架构的内在逻辑和部署策略。

归根结底，所有产品和架构都必须遵循分布式计算和物理学的基本原理：运行应用和产品的计算机性能有限，通过物理连接和逻辑协议实现的通信必然有延时。因此，应该承认架构才是影响应用性能和可伸缩性的决定因素。性能参数是无法简单地通过更换软件，或者“调优”底层软件架构来改善的，我们必须在架构的设计（或重新设计）上投入更多精力。

作者简介：

兰迪·斯塔福德拥有 20 年从事软件行业的经验，他身兼多职，既是开发者、分析师、架构师，又是经理、顾问、作家/投稿人。目前他是甲骨文公司中间件精英团队的成员，负责帮助全球范围内的客户和组织验证概念项目，审查架构，解决生产问题，他致力于研究网格、SOA、性能优化、高可用性，以及 JEE/ORM。

分析客户需求背后的意义

Seek the Value in Requested Capabilities

埃纳尔·兰德雷（Einar Landre）

顾客和最终用户通常提出的所谓需求，只是他们心目中可行的解决方案，并不是问题唯一的解决途径。F-16“战隼”（战斗机）的设计师哈里·希拉克尔（Harry Hillaker）曾就此给出过一个经典的案例。军方最初对F-16 的设计需求是：飞行速度在 2~2.5 马赫 （译注 1）之间的低成本轻型战斗机。要知道当飞行速度由 1 倍音速变为 2 倍音速时，空气阻力会增至原来的 4 倍，考虑到因此所需要的动力，及其对机身重量的苛刻条件，满足这样的需求，即使是在今天也绝非易事。

设计团队追问军方为什么需要 2~2.5 马赫的速度。答复是“为了迅速撤离战场”。了解真正的需求后，设计团队对症下药提出了有效的解决方案：通过提升推力重量比（Thrust-to-weight Ratio），改善战斗机的加速性能和机动性能。用灵巧性取代了最初对速度的需求。

软件开发也应该借鉴这条经验。架构师可以通过询问客户，分析客户要求的功能和需求的真正意义，定位真正的问题，从而提出比客户的建议更好、成本更低的解决方案。通过关注问题的真正含义，理顺需求的轻重缓急：把最有价值的需求摆在第一位。

译注 1：马赫（Mach）是速度单位，1 马赫约等于 340 米/秒，即音速。

该怎么做呢？敏捷宣言提供了答案：“客户合作重于合同谈判（Collaboration over contract）”。具体来说，架构师应该通过与客户面对面的交流，关注客户的需求，引导客户回答“为什么”的问题。要知道说明“为什么”并不容易，因为客户往往觉得问题是不言而喻的。此外，避免与客户讨论技术上的具体实现，这样做会喧宾夺主，因为此时应该关注的是客户的问题，而不是软件开发的问题。

作者简介：

埃纳尔·兰德雷（Einar Landre）25 年来一直从事着与软件相关的工作，他当过程序员、架构师、经理人和顾问，还写书和主持会议。目前，埃纳尔正忙于 StatoilHydro 公司的企业应用服务项目，他参与开发关键业务应用，评审架构，以及改进开发过程。他擅长面向服务的架构（SOA），领域驱动设计（domain-driven design）和利用多代理结构设计大规模软件密集型的联网系统。

起立发言

Stand up!

乌迪·大汉（Udi Dahan）

许多架构师都是从技术岗位上成长起来的，他们擅长与机器打交道。然而架构师更需要与人打交道，无论是劝说开发人员接受具体的设计模式，还是向管理层解释购买中间件的利弊，沟通都是达成目标的核心技能。

虽然架构师对项目的影响很难界定，但有一点是清楚的：无论架构师的建议多么正确，如果开发人员和管理层都不买账，架构师就无法取得事业上的成功。有经验的架构师都很重视“推销”自己的想法，也明白有效沟通的重要性。

介绍人际沟通诀窍的书不少，我只想向大家介绍一个简单实用的技巧，它可以显著地改善沟通效果，让大家的工作更上一层楼。在两人以上的场合发表意见时，请站起来。无论是正式的设计审查，还是借助图表进行非正式的讨论，起立发言非常重要，尤其是当其他人坐着的时候。

当你站立时，无形中增添了一种权威和自信，自然就控制了场面。听众不会轻易打断你的发言。这些都会让你的发言效果大为改观。

你会发现，站立时可以更好地利用双手和肢体语言。在十人以上的场合，起立发言方便你与每位听众保持视线接触。眼神交流、肢体语言等表达方式在沟通中的作用不可小觑。起立发言还可以让你更好地控制语气、语调、语速和嗓门，让你的声音传得更远。当你讲到重点内容时，注意放慢语速。发声技巧也能显著改善沟通效果。

让沟通事半功倍，起立发言是最简单、有效的方法！

作者简介：

乌迪·大汉是位“软件精简主义者（The Software Simplist）”。他凭借在解决方案架构（Solutions Architecture）方面的造诣，连续三年被微软授予令人羡慕的 MVP 称号。乌迪是微软的 WCF、WF 和 Oslo 技术顾问，他也是微软软件工厂计划咨询委员会的成员，参加了微软模式与实践（Patterns & Practices）组的 Prism 项目。他在全球各地提供培训、指导和高端架构咨询服务，尤其擅长设计面向服务的、可伸缩的、安全的.NET 架构。

故障终究会发生

Everything Will Ultimately Fail

迈克尔·尼加德（Michael Nygard）

硬件会出错，于是我们增加冗余资源来提升系统的可靠性。这样做虽然可以避免由于单点故障引起的系统错误，但同时也增加了至少有一台设备出错的概率。

软件会出错，由软件构成的应用程序自然也会出错，于是我们增加额外的监控程序，好在应用失效时报警。但是监控程序也是软件，一样会出错。

人无完人，我们也会犯错，所以我们把操作、诊断和处理都变成自动化。可是自动化虽然降低了主动犯错的概率，却增加了错误被忽略的概率。何况任何自动化系统应付环境变化的能力都比不上人类。

于是我们又为自动化增加监控，结果是更多的软件，导致更高的故障率。

计算机网络由硬件、软件和长距离的线路构成，当然也会出错。实际上，即便网络工作正常，由于状态空间的无限性，网络的行为也是不可预测的。独立部件的行为可以确定，但是所有部件组合起来，就会无法避免地出现混沌现象。

所有用来避免故障的安全机制都会带来新形式的故障。集群软件可以把应用程序从故障服务器转移到正常的服务器，但如果集群网络功能失常，又会出现“网络分区综合症（split-brain syndrome）”（译注 1）。

译注 1：split-brain syndrome 原意是裂脑综合症，指两个脑半球的感觉及运动功能的连接被切断，以致患者丧失日常生活自理能力的病症。这里用来比喻网络分区之间正常通信失效的情况。

别忘了三哩岛核电站泄漏事故（Three Mile Island accident）（译注 2）是由于减压阀引起的，而减压阀本身是用来避免过压故障的安全机制。

既然系统必然会出错，我们该怎么办呢？

应当承认系统中必然存在着不同形式的故障隐患，无论如何都无法彻底消灭。如果你否认这个事实，管理故障和限制故障就无从谈起。只有承认这一点，才能针对特定的故障设计对策，正如汽车工程师知道交通事故无法避免，所以设计撞击缓冲区（crumple zones）来保护乘客一样，你也可以设计预防措施来限制故障，保护系统其余部分。

如果不事先设计好防范故障的模型，就无法应对威胁系统安全的意外情况。

作者简介：

迈克尔·尼加德著有《Release It! Design and Deploy Production-Ready Software》（Pragmatic Bookshelf 出版社），该书荣获 2008 年 Jolt 生产力大奖（Jolt Productivity award）。他的博客（*http://www.michaelnygard.com/blog*）上有更多文章可供阅读。

译注 2：Three Mile Island accident 指 1979 年发生在美国宾夕法尼亚州三哩岛的核电站泄漏事故。事故最初由二号反应堆的辅助回路冷凝水泵故障引起，导致二号反应堆内温度和压力上升，触发堆内的减压阀开启。卸压后，由于减压阀故障，阀门未能按预期自动关闭，进一步导致冷却水大量溢出，堆心温度上升。待工作人员发现问题所在的时候，47%的堆心燃料已经融毁并发生泄漏，幸好有防护外壳阻挡，未造成人员死亡，但经济损失超过 10 亿美元。

我们常常忽略了自己在谈判

You're Negotiating More Often Than You Think

迈克尔·尼加德（Michael Nygard）

我们都面临过削减预算的要求。如果资金运转捉襟见肘，技术方案只能委曲求全。比如下面的情景：

“我们真的需要这东西吗？”项目投资人发难道。

要知道“这东西”很可能是系统运行必不可少的条件，比如：软件许可证、冗余服务器、离站备份系统，甚至供电设备。更可气的是，投资人总是一副教训的口吻，仿佛只有他懂得钱要花在刀刃上，我们都是毛头小子，就会浪费钱买漫画书和泡泡糖。

答案明摆着是“真的需要”，但很少有人回答得这么干脆。

毕竟我们都是工程师出身，搞工程就是要统筹全局、权衡利弊、适当妥协。例如我们知道只要为数据中心配备发电机，再招几个廉价的实习生，就不必购买奢侈的供电设备。所以我们不会干脆地说“真的需要”，多半会这样回答：“好吧，不买第二台服务器也行，只不过在对系统进行例行维护时必须停机。另外，奇偶校验位翻转也会导致系统崩溃，不过这个问题可以通过购买带奇偶校验的内存来解决。剩下来我们就只用对付操作系统的崩溃了，操作系统大约 3.9 天崩溃一次，解决办法是每晚重启一次系统。当然，这项工作可以找实习生来做。”

虽然句句属实，但绝对是对牛弹琴。投资人想听的只是“好吧”两个字，对后面的话毫无兴趣。

问题出在你没有认清自己的角色，你还是把自己当成工程师，而项目投资人明白他在跟你谈判。工程师总是想尽办法寻求合作，谈判者则绞尽脑汁占得先机。谈判时，绝不能在对方的第一个要求上妥协让步。其实，我们应该这样回答“真的需要吗”这类问题：

“单台服务器每天至少会崩溃三次，系统负载加重的话情况会更严重，没有第二台服务器，我们无法保证你给董事会做演示时一切正常。说实话，我们至少需要四台服务器，两个一组构成两组高可用服务器组，这样可以在需要时断开其中一组，而不必被迫关闭系统。即使剩下的一组中又有一台服务器意外崩溃，也不会影响系统的正常运行。”

当然你并不需要第三台或第四台服务器，这只是谈判的策略，好把对方的话题引开，强调你已经勉力而为，如履薄冰，然后提出更多要求。顺便提一下，如果你真的得到了两台额外的服务器，一台可以用来做产品的质量保证（QA），另一台可以留着随时备用。

作者简介见第 17 页。

量化需求

Quantify

基思·布雷思韦特（Keith Braithwaite）

“速度快”不能算作需求，“响应灵敏”和“可扩展”也不能算需求，因为我们无法客观地判断是否满足了这样的条件。但这些又确实是用户想要的。架构师的工作在很大程度上是要平衡这些需求之间的不可避免的冲突和矛盾，同时又使系统尽可能地满足它们。如果缺乏客观的标准，架构师就只能任凭挑剔的用户和偏执的程序员摆布（“还不够快，我拒绝接受”和“还不够快，我不能发布”是他们的口头禅）。

在记录需求的过程中，经常会出现模糊的描述，比如“灵活”、“可维护”等。实践证明，所有这些描述都可以量化，并设定相应的检测标准（甚至连“易于使用”这样的需求也可以量化，只是要费些工夫）。没有量化的需求会导致用户验收系统时缺乏依据，架构师不知所措，开发工作失去正确的指导。

我们可以通过一些简单的问题来量化需求，例如：数量有多少？在什么阶段？有多频繁？不能超过多长时间？增加还是减少？占多大比例？如果得不到答案，需求就不明确。这些答案应该包含在系统的业务策划方案里，否则还得费不少脑筋。如果架构师无法从业务部门那里得到这些数字，应该先反思原因，然后再想办法。下次再有人对系统提出“可伸缩（scalable）”的要求，一定要问清楚新用户从何而来，为什么数量会增加，以及何时增加，会增加多少。拒绝“很多”、“马上”这类模棱两可的答案。

对那些无法明确量化的指标也必须给出一个描述范围，比如：上限、下限等。没有范围，就没办法理解具体的需求。随着架构的发展，这些指标范围会用来检查系统是否（仍然）符合要求，久而久之，性能指标的变化会反馈出有价值的信息。不过，要找出这些指标范围，并根据它们来检验系统，是件既费时又花钱的事。如果客户不关心系统的性能表现（既没有需求文档，也没有口头要求），也不愿意为性能测试买单，那么性能对他们来说可能不重要，这时你可以从容地将精力放到其他值得关注的系统问题上去。

正确的需求描述应该像这样：“必须在 1500 毫秒内响应用户的输入。在正常负载（定义如下……）的情况下，平均响应时间必须控制在 750~1250 毫秒之间。由于用户无法识别 500 毫秒以内的响应，所以我们没必要将响应时间降低到这个范围以下。”

作者简介：

基思·布雷思韦特 1996 年开始写软件赚钱，在此之前他只是个业余爱好者。他的第一份工作是维护用 lex 和 yacc 开发的编译器，此后他设计过 GSM 网络的微波传播模型，用 C++解决过航空运输由于季节性需求变化引起的资费调整问题。随后他从事咨询工作，接触到 CORBA、EJB 和电子商务。目前他是 Zhhlke 公司的首席顾问，管理公司的敏捷实践中心。

一行代码比五百行架构说明更有价值

One Line of Working Code Is Worth 500 of Specification

艾利森·兰德尔（Allison Randal）

设计拥有无穷的魅力。我们运用系统的方法，详细地描述问题空间（problem space），审视解决方案，找出缺陷和可以完善的部分，获得的效果有时令人拍案叫绝。架构说明书（specifications）很重要，因为它描述了构建系统的模式。但是静下心来全面彻底地理解架构——既从宏观上把握组件之间的交互，又着眼于组件内部的代码细节——也很重要。

不幸的是，架构师往往容易被抽象的架构所吸引，沉迷于设计过程。事实上，仅有架构说明书是远远不够的。软件项目的最终目标是建立生产体系（production system），架构师必须时刻关注这个目标，牢记设计只是达成目标的手段，不是目标。摩天大楼的建筑师如果一味追求美观而无视物理定律，迟早会自食苦果。我们的目标是可工作的代码，对软件项目而言，忽略这一点就是灾难。

应该重视团队成员的意见，是他们在实现你的设计。要善于倾听，如果大家对设计提出疑问，很可能设计确实存在问题，或者不够清晰。这时架构师应该与团队成员合作，共同作出决策，修改设计以符合实际情况。没有天生完美的设计，所有的设计都要在实现的过程中逐步完善。

如果你亲自参与开发，应该珍视自己花在写代码上的时间，千万别听信这会分散架构师精力的说法。参与项目所付出的努力，既能拓展你的宏观视野，也能丰富你的微观视界。

作者简介：

艾利森·兰德尔是开源项目 Parrot 的首席架构师，兼开发组长。在长达 25 年的程序员生涯里，她开发过各种软件，包括游戏、语言分析工具、电子商务网站、编译器、数据库复制系统等；她设计过编程语言，组织过技术交流会，做过项目经理、编辑和顾问。此外还担任过开源软件组织的主席，编写过两本书，并且成立了一家技术出版公司。

不存在放之四海皆准的解决方案

There Is No One-Size-Fits-All Solution

兰迪·斯塔福德（Randy Stafford）

架构师应该坚持培养和训练“情境意识”（contextual sense）——因为我们遇到的问题千差万别，不存在放之四海而皆准的解决方案。

“情境意识”这个贴切的说法，最早由埃贝哈特·雷克廷（Eberhardt Rechtin）在《Systems Architecting: Creating & Building Complex Systems 》（Prentice Hall 出版社）一书中提出，并予以深刻阐述：

> [运用“试探法”设计复杂系统架构的要点是]调查有经验的架构师处理复杂问题的方式。有经验的架构师和设计师的答案如出一辙：只须使用常识……[一个]比“常识”更贴切的说法是“情境意识”——在给定情境下对合理性的把握。架构师通过学习和实践，不断积累的案例和经验，建立足够的情境意识。他们通常需要十年的磨练，才能解决系统层次的问题。

在我看来，软件行业的一个大问题，是那些负责解决问题的人积累的情境意识不够。毕竟软件行业起步不过六十多年，并仍在飞速发展。当这个问题消失时，也许就标志着软件行业成熟了。

我做咨询工作时频繁碰到这类问题。典型的情况包括：该用领域驱动设计（注 1）时没有使用；软件方案设计过度，偏离了实用性和眼前的基本需求；或者在解决性能问题时，提出的建议不合理，甚至毫不相关。

掌握软件开发模式的重点在于拿捏应用的时机，这也是分析问题时避免胡乱猜测和矫枉过正的关键。显然，无论是设计系统架构还是分析问题，都不存在万能钥匙，架构师必须培养和训练情境意识，才能更好地设计架构和解决问题。

作者简介见第 11 页。

注 1：见埃里克·埃文斯（Eric Evans）的著作《Domain-Driven Design: Tackling Complexity in the Heart of Software》（Addison-Wesley Professional 出版社）。