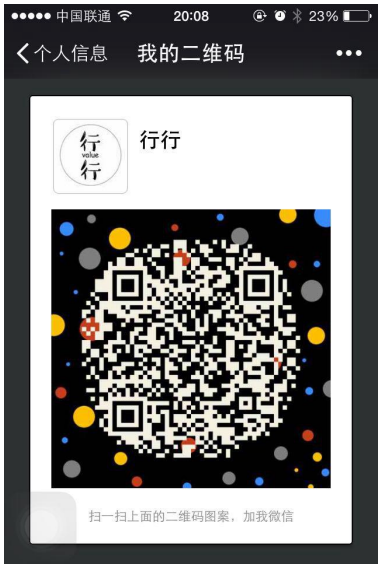
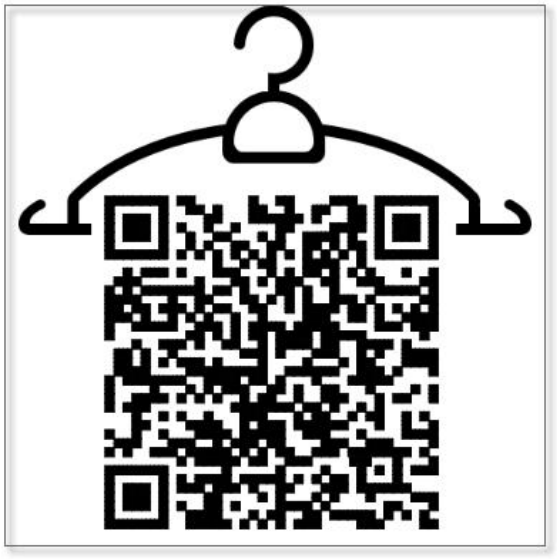
1、小编希望和所有热爱生活，追求卓越的人成为朋友，小编：QQ 和微信 491256034 备注书友！小编有 300 多万册电子书。您也可以在微信上呼唤我　放心，绝对不是微商，看我以前发的朋友圈，你就能看得出来的　。

2、扫面下方二维码，关注我的公众号，回复电子书，既可以看到我这里的书单，回复对应的数字，我就能发给你，小编每天都往里更新 10 本左右，如果没有你想要的书籍，你给我留言，我在单独的发给你。



扫此二维码加我微信好友



扫此二维码，添加我的微信公众号，查看我的书单

周爱民（Aimingoo），国内软件开发界资深软件工程师，架构师。有十余年的软件开发、项目管理、团队建设的经验，曾任盛大网络平台架构师、支付宝业务架构师等职位。著有《Delphi 源代码分析》、《大道至简》和《JavaScript 语言精髓与编程实践》等专著。

图书在版编目（CIP）数据

大道至易：实践者的思想 / 周爱民著. -- 北京：人民邮电出版社，2012.6

（图灵原创）

ISBN 978-7-115-28217-0

Ⅰ. ①大… Ⅱ. ①周… Ⅲ. ①软件开发—项目管理 Ⅳ. ①TP311.52

中国版本图书馆 CIP 数据核字（2012）第 103852 号

内 容 提 要

本书可以看成是《大道至简》的姊妹篇，是以软件工程为体系、以组织结构为视角，融合系统架构师、项目管理者和软件开发人员三种角色实践的思想总成。本书讨论这些思想，并陈述它们所基于的原则、背景与获得过程。

本书适合各类工程管理人员、软件开发人员和架构师阅读。

图灵原创

大道至易——实践者的思想

* * *

◆ 著　　周爱民

责任编辑　杨海玲　李松峰

◆ 人民邮电出版社出版发行　北京市崇文区夕照寺街 14 号

邮编　100061　　电子邮件　315@ptpress.com.cn

网址　http://www.ptpress.com.cn

北京　　印刷

◆ 开本：800×1000　1/16

印张：25

字数：519 千字　2012 年 6 月第 1 版

印数：1-5000 册　2012 年 6 月北京第 1 次印刷

ISBN 978-7-115-28217-0

定价：69.00 元

读者服务热线：（010）51095186 转 604　印装质量热线：（010）67129223

反盗版热线：（010）67171154

与架构师同行（序）

我认识周爱民多年，本书是他的第五部作品，也是“周爱民风格”最浓烈的一部。作者把“冥想”式的写作发挥到了新的高度，任何一个小小的、普通人熟视无睹的问题、现象、矛盾，都能引起作者滔滔江水连绵不绝的思考，追根溯源的论辩和富有禅机的跳跃的联想。这是周爱民思维的特点所在，他能够长时间连续不断地思考一个问题，越挖越深，既不知疲倦，也不向现实妥协。这是一种独特的思维品质，是如我一样普通人所难以企及的，也是这个时代所稀缺的。这种独特的思维品质，既是这本书价值的源泉，也成为对读者，以及我这个受托作序的非技术人来说最大的挑战。所以在这篇序言里，我不敢说能把握这本书的思想实质，但会坦率地谈谈我对书中所涉及的一些问题的个人看法。

首先是关于这本书的主题和内容。这本书当然是在谈软件开发的问题，但涉及的方面又不仅仅是开发，甚至可以说，作者主要不是在谈开发，而是在谈组织。而且他不是在谈某一种具体的组织，不是代码的组织、构件的组织、架构的组织，甚至也不是工程的组织和人的组织，他是在整体上来思考“组织”这件事情的一般原则，然后用其所思所得来“格”代码、“格”架构、“格”工程、“格”人的组织。从书中，读者可以跟着他的思路去体会他是如何“格物致知”，然后又反过来“致知格物”。

为什么要把组织问题放在中心位置去探讨？因为良好的组织是解决软件规模问题的唯一希望。而随着软件行业的逐渐成熟，组织和架构已经取代编码而成为软件开发的核心问题。

十几年来软件开发行业发生了一个重大的变化，那就是软件开发已经渐渐演变成了一个以配置为主、编程为辅的过程。这个观点最早是 20 世纪 90 年代中期由著名的 C++网络编程框架 ACE 的创造者 Doug Schmidt 教授提出来的。当时一系列重要的基础性的标准，如 C 语言、TCP/IP 协议、POSIX 标准已经奠定，Schmidt 教授意识到，一旦一小部分人开发出足够多的、高质量的、可配置的核心软件，其他人就没有必要重复发明轮子，而是可以站在他们的肩膀上，通过配置和扩展来完成任务。尽管配置的过程本身还是通过编码的方式来完成的，但本质上这些代码只是在定制核心软件的运作方式，它们的技术难度比编写高质量核心软件至少要低一个数量级。

这个观点被历史证明是非常具有远见的，特别是随着 Java 等跨平台语言的风行、开源运动的兴起和 Web 的成功，绝大多数不同种类的应用都在共享相同的基本特征，这样也就出现了一大批门类齐全、质量过硬的核心知识资产，它们以基础软件、中间件、框架、程序库和开源代码的方式存在，可以被以各种方式复用。在这些资产之上，程序员社群形成了有效的、开放的知识分享机制，从而使得一般性的软件开发变得相对容易，而将软件开发这件事情的挑战上移到架构层面，或者更具体地说，如何有效地分配职责、组织资源、建立资源节点之间的关系和契约，已经成为大规模软件开发的根本问题。

这就是本书的发力点。作者正是站在架构师的高度上来看待软件开发，甚至更大的意义上，看待以软件开发能力为核心的企业在产品战略、人力配置和文化方面的组织问题。作者在过去几年里，先后在两家超大规模互联网企业中做架构师，他在其中的矛盾、迷惘、挣扎和成就是这本书的思想源泉，而他对于架构师角色的实践和领悟，也是这本书的价值所在。作者认为，软件开发发展到今天这个阶段，架构问题是真正的方向性问题，架构之争是真正的路线斗争，架构错误是唯一致命的、不可挽回的错误。只要架构适用，其他的错误都是局部性的、可以控制的。一旦架构发生方向性错误，无论是人的架构，还是软件的架构，其代价可能就是一个项目甚至一个企业组织的生命。因此，这本书通篇都在讲架构问题，讲组织问题，讲规模问题。读这本书的时候，如果能够始终记住这一点，那么理解起来会容易一些。

看上去这本书的四个篇章，彼此相对独立，谁也不挨着谁。但实际上，贯穿全书的主轴，就是从宏观的组织关系上和微观的角色职责上，在不同的领域里反复谈组织架构问题。在这个过程中，作者也提出了一些鲜明而有趣的想法，比如让架构师成为开发团队中的核心角色，甚至把人员和项目管理这样传统强势的角色也仅仅视为架构师的支撑角色。作者的核心观点，说到底很朴素，无论是在人员组织还是技术架构上，作者提倡各就各位、各司其责、不增不减、不垢不净的风格，尽量简化由于异动而导致的非本质复杂度。这些观点，至少成一家之言，值得思考。

当然，用这种“中心思想”、“段落大意”式的语文课方式来归纳这本书，是非常不合适的，因为这本书的写作风格确实是非常独特。以周爱民的水平和资历，他大可以像很多流行书作者一样，居高临下，弄出几条十几条原则，以一种非常具有“架构感”的组织方式来组织这本书的内容本身。但是如果架构问题能够以这种方式讲清楚的话，那它早就不是问题了。如作者所说，架构本身是“以序容易”，是为了约束系统自由度，从而使架构之外的决策空间更小、更容易，那么架构本身的设计，则成为重点和难点。如何成为好的架构师，如何教授和交流架构设计的经验和思想，迄今仍然不能说找到了办法。作者采取的方式，就是老老实实把自己的思维过程和喃喃自语都很原生态地记录下来，从而形成一本“心语”之作。因此，如果读者也能够“心心相印”，潜下心来仔细品读这本书，那么就相当于跟作者一起进行了

一段旅行对话，或者，一起在思维的教武场里杀个几进几出。或许这种如切如磋、如琢如磨、反复捶打、反复锻造的风格，倒确实能够帮助有心的读者获得一些架构设计的经验。

因此，无论从内容上，还是从行文风格上，这本书都是非常独特的，也有其独特的价值。作者的作品，一向是长销之作，其价值也往往需要一段时间才能被真正认识，我不敢说自己能够在短暂的阅读当中充分理解这本书，但是我相信其中所蕴藏的价值和诚意。

当然，无论对于书的观点、内容还是展现形式，我也有自己的看法，其中也有不同意见。

比如在行文上，我认为作者对于读者设置了比较高的理解门槛，要求读者必须跟着他的文字和思绪流动。如果读者以当下流行的速读式阅读、跳跃式阅读甚至微博式阅读的方式来读这本书，我敢说不但不可能把握其中的精华，甚至连一些基本概念可能都会搞错。

再比如在内容的铺述方面，上至人员组织架构，下至程序设计语言中一些架构性的设计，在一本 300 多页的书里跨越如此巨大的领域，我相信是前无古人。尽管作者确实始终以一主线贯穿，但是能够洞解其中滋味的读者，恐怕也是少之又少。好在读一本书，也并不是要求全部理解，懂一点，学一点，常读常新，也是一种不错的体验。

至于作者的思想，我能够置喙之处不多。勉强要说一点，可能就是一个搭配问题。从组织、人员、文化到技术、编码，既然每个层面上都有自己的架构问题，都有若干不同的架构选择，那么可能不同层次之间架构选择的搭配问题，或许也应该是这本书应该触及的。一个优秀团队的人员组织架构，必须与其行业领域和产品项目类型彼此搭配，也必须与其选择的技术框架和架构彼此搭配，也必须与其软件过程的管理方法彼此搭配。抽象地在每一个层面上去谈什么是好、什么是坏，是没有意义的。CMMI 的模型是个好的过程架构，但是谷歌和 Facebook 决不会采纳，因为这个架构与其他领域的既定架构不兼容。什么样的组织架构与什么样的技术架构搭配是合适的，这本身是一个非常有趣和有意义的问题，可惜这本书落墨不多，不得不说是一个可以改进的地方。

总的来说，这是一本需要认真读的书，也是一本值得认真读的书。它的价值需要读者与作者在思维互动中逐渐体现。如果你是这样的读者，我会愿意向你推荐这本书。而如果你并不喜欢深彻的冥想式的阅读和思考，那么你很难从这本书中获得太多。

孟岩

写给读者

这不是一本能够速读的书。

如果您读过《大道至简》，那么本书会完全颠覆您此前的阅读体验。尽管它们可以看成姊妹篇，并且在内容上也有承接，但却有着完全不同的写作方法与思维系统。又如果您了解传统含义上的“软件工程”，那么也请您先放弃固有的观念。本书所述的，在您理解之后尽可以同样放弃：所有接近与远离真理的，都必被打破。

这是一本面向多类读者的书。

本书是“实践者的思想”，并没有确指是“软件工程实践”。因而所言的是以软件工程为体系、以组织结构为视角的，在系统架构、项目管理和软件开发三种角色实践中的思想之总成。如果读者是其中的某一角色，建议从相应的分篇入手阅读；如果读者同时兼有多种角色，并有丰富的实践经验，可以从总论入手阅读。

这是一本讨论思想方法的书。

如同《大道至简》一样，本书不讨论具体的做事方法。所有行事的方法与准则，都不能孤立于其背景而存在，因此方法是死的，得到这些方法的思想是活的。但亦如行事的方法一般，思想的方法也不能脱离其背景存在。所以本书讨论这些思想，并陈述它们所基于的原则、背景与获得的过程。

这是一个视角而非全景。

无论如何，我们只能触及事物全体的一个或几个侧像。以管窥豹并不可笑，可笑的是每个人都认为他看到了豹的全部。认识到我们自己的眼界之狭隘，比得到这个狭隘认识中一个确实的影像远为重要，前者是进步之始，后者则是自满之源了。本书所讨论的工程模型仅是软件工程的一个视角，只是我将它当成软件工程的本质问题来讨论，毕竟无论何种工程，本质上仍然是人的行为，而不是事的流程。

前　言

易是变化

台湾的高焕堂先生曾说架构的要旨是“以序容易”，我解释成“用规则来包容变化”，高老师说很合他的本意。这里的“易”，指的就是变化。既是变化，那当然是艰难而复杂的了。然而我们通常说一件事易做或另一件事不易做，这里的“易”却都是指简单的意思。所以“易”既是无穷的复杂，也是至极的简单，关键在于如何“容”它。

我们得看清什么是变化。

盲人摸象是一个很好的故事，因为每一个盲人的认识都是其固见的、自见的，以及自证的真理。然而盲人们的观点放在一起的时候，却是一个笑话。因为看笑话的人看到了“变化”，而这些局外者原本认为那头大象是不变的、确定的、唯只一个形象的。同样，面对任何我们所见的变化，或对变化中的任何一个认识，我们都认为那不过是笑话。再深彻地透视这一点，其根本在于：我们也有一个对大象的“认识”，只因为这一认识看起来——或我们认为——比那些盲人更为高明、正确，所以我们才别出了盲人，看到了笑话。

正是我们对一个事物固有的认识，制造了盲人与盲人的笑话。

倘若我们的认识是不可易变的，那么我们今天就已经看到了真理，看透了世事万象，我们已成至人，故而我们不需要存在亦无需进化：一切于我们而言，必须静止；一切于我们的认识而言，不可复加；一切于我们的思维而言，不可偏侧。而这，看起来不正是我们自盲了双目，自演了笑话吗？

反过来，我们得承认变化的存在。我们所要做的，只是承认自己是一个盲人，可以从一个方面去触及这一事物，形成一个认识，用一些规则、规格或概念去确指它。用同样的方法，我们触及这个事物的方方面面，进而得到与这个事物最为接近的一个全像，这就是我们关于这一事物所有的、而又未尽的知识。

知音变而得律，有容则易。

简也不是简单

这本书是《大道至简》（以下称“《简》”）的续作，那我又是怎么认识《简》的呢？究我们的认识的根底来说，我们向来不是缺乏思考的。我们有着种种在实践中的选择，这些选择是基于一些认识的，而又产生了一些新的认识。而我所有的醒觉，并不来源于对后一种认识的反复持续，反而在于对前一种——既有的认识的观察。

《简》一书的源起，正是一次对“我在干什么”的发问。在这一反思中，我最终找到了我在工程中的种种位置，并将这些位置在我既有的知识中一一标点。然后，我们就看到了那幅 EHM（工程层状模型）图（参见“总论”图 0-1）。

追究起来，EHM 图以及它所代表的我的认识已经是 8 年前的事情了；完成《简》一书，也已经是 6 年前的事情了。6 年前，我将《简》作为电子版发布的时候，也同时得到了一个新的职务：架构师。然而如果你仔细地观察 EHM 图，就会发现它根本上是不能容纳“架构”这个角色与职责的。

“我在干什么”，仍然是我这 6 年来最主要的思考。事实上，这也促使我抬起头来，观察周围营营碌碌的人们，了解他们在做什么，在围绕什么去做，以及——也是最重要的是：他们得到了什么。如同《简》中所说的愚公一家，数百年挖山不止，最终却不得不筑关自守，我们是不是也一样背离了初衷，成为一个不知所为的愚者？

《简》事实上是打开了我思维的一扇门，让我看到“我既有的那些东西”是何等粗糙，又曾经是如何粗暴地塞入了我的头脑。例如我在计算机专业中所学的第一门功课是《数据结构》

(1)，这本书的第一页便写着：“用计算机解决具体问题，首先要加以抽象。”但十余年来，我从未思考过“什么是抽象”，又或者我的思维以及由此驱动着的软件开发活动中“何处是抽象”。抽象被作为我们——程序员的一种事实的本能，或被驯育的技能塞入了我们的举手投足，控制着我们而又让我们无知无觉。

《简》在这“无知无觉”的宁静中轻一叩击，“叮”的一声，我的眼前便亮了。观之简，便在其原始的粗鄙；思之得，便在行道中的一凛。

没有答案

《简》一书只提出了现象，而无有答案。2009 年在北京的 QCon 大会中，我跟几位朋友谈到过这个问题。我坦承这一事实：若《简》尚有一点可读处，便在于它粗暴而又挑衅地把种种问题摆在我们的面前。但对许多人来说，这毫无益处。因为他们或是已经看了这些，或已经为之麻木，或是在贪求某种法子来逃避这些东西带来的影响。

总而言之，《简》是“没有答案”的。我彼时也是“没有答案”的，我只是觉醒过来，并看到了一些事实上存在已久的东西而已。与此相同，《大道至易》这本书也“没有答案”。尽管我提出了一些貌似答案的东西，但我确实是在保守地、谨慎地谈论它们，并且时时地（譬如现在）警示读者：这是在某些背景下的一时之选。

“讲述某些我知道或想到的答案”是本书中最没有价值的部分，即使它们存在，即使看起来“非常有用”，也仅是一时的表象而已。

与《简》不同的是，《大道至易》提出了种种问题。就我的认识来说，现象与问题之间是有着一个巨大的跨越的。例如《简》中提到的“跑不动了”，它可以成为一种催人奋进的现象，也可以是脱水这样的问题。作为旁观者，若只见现象，便只会责骂鞭策；若能见问题，则知道“一杯冰水”才是解决之道。在我看来，正确的问题与“解”之间的距离是可显见的，而一个表面的现象或错误的问题，就离真正的解淼远而无际了。

《大道至易》是一本提出问题的书。

不设节名

正因为问题重要，而答案不重要，所以我认为给每段内容加以一个“有诱惑力的标题”是多余的。因此本书分篇、章、节三级目录，但节下的小节只有“一、二……九”这样的中文标号，没有标题，是谓“不设节名”。

事实上在我撰写的过程中，这些小节多数是有标题的。但最终我发现，若是有一个标题，则读者便非常容易陷入这个标题的“含义”之中，以之作为小节之纲要、主旨或主张。而这又正是我所不愿意的，因为我原本只是拿这些标题来搭个架子而已。

也就是说，本书原本是只有骨架，而没有血肉的。真正有血肉的，是你面对它的种种思考：那些没有名字的小节，是可以由你去补个标题的；那些没有定论的观见，是可以由你去设定的。我要做的，只是把问题写出来，把我的想法写出来，把我的观见写出来。这些，既不是

真理，也不是我自作了固见的认识。

我之于盲人，相距并不甚远，只是我惊觉我的盲目，故而对周围多做了些探索；又或者我还能觉出蒙眼的黑布才是真实的问题，揭去之后，便一切历历在目、观之若即了。

只不过一刹那，定是相当刺目的。

不为读者

对于许多读者而言，本书的“总论”是一个过高的阅读起点。因此本书的编审老师曾建议将“总论”移至最末一篇。为此我犹豫了很久，并也曾数次动摇。最终，我仍坚持将它作为本书的开篇。尽管编审在来信中写道：“根据我们的经验，好的开头对销量的影响可能达到三成。”

这真的是不小的诱惑。

相比来说，本书的第一篇“似乎”更适合作为“好的开头”——这也确实是编审老师们所推荐的。然而，事实上这一篇我是最晚成稿，写得最累，以及认为它是最不易读的。尽管表面来看，读者可以从这里轻松入手读上几页，然后觉得这是本“还有点趣”的书。然而它到底写的是什么呢？

离开总论的框架，所谓第一篇，不过是可以换得你的同感的抱怨而已。而真相，绝非如此。就写书出书而言，我当然可以用读者喜欢的方式去写作，并用读者喜欢的方式去包装那些篇章，然而这样一来，读者便会在“一开始”就失去整体的、系统性的认识，并认为这本书不过是又一本讲述软件工程的《大道至简》。

我可以为了“读者的口味”而改变这本书的著述体系吗？这是我最终的设问。而我很快地给出了答案：不！

总论是本书所述思想的核心，而后续几篇则是这一思想下的几个视角。让读者抛开总论而去窥见某个视角，与盲其目使之摸象何异？与予其管使之窥豹何异？我在前言中所反对的思想与方法，何以成了我去争取“读者人数”与迎合“读者口味”的手段？！

因此写下这段文字，以此自勉：莫因读者“买书如买纸”，便将书作了“字纸”去写，又或将书作了“纸页”去卖。

既如此，便仍让总论做了第一篇罢。这样一来，即便读者从它处读起，也总还能时时记得：有一篇“总论”在前头，或许蕴含着作者思想的主旨。

致谢

感谢所有的读者、编者以及一直以来支持我的朋友们。

感谢孟岩先生为本书作序。

感谢图灵出版公司。

感谢我的小学数学老师。

感谢 Joy。En，……我最爱的妻。

————————————————————————

(1) 《数据结构》，严蔚敏、吴伟民编著。

目　　录

与架构师同行（序）

写给读者

前　　言

总论　领域角色的关注

第一节　什么是领域角色的关注

第二节　基于组织视角的观察

第三节　工程的本质问题是组织

第一篇　具体工程下的组织与行为

引言　管理中的逻辑

第一章　任人治事：组织行为的基本认知

第一节　刺秦与灭秦

第二节　看到别人能做什么

第三节　要做事，不要管理

第四节　伯夷与叔齐是怎么死的

第二章　谋定后动：项目的存在权

第一节　试错通常是无能的托辞

第二节　合法的山大王为什么没能成功

第三节　自己想办法

第三章　具体而微：工程是系统而不是事

第一节　做事的选择

第二节　你要什么

附录一　行在道上，从局部到全局

附录二　本来面目——大教堂、集市，与作坊

附录三　杀不死的人狼——我读《人月神话》

第二篇　程序源流：从计算到系统（上）

引言　简单的本源

第四章　计算系统

第一节　数，以及对数据的性质的思考

第二节　逻辑

第三节　抽象

第五章　语言及其面临的系统

第一节　语言

第二节　从功能到系统

第六章　程序设计的核心思想

第一节　数据结构：顺序存储

第二节　数据结构：散列存储

第三节　执行体及其执行过程中的环境

第四节　语法树及其执行过程

第五节　对象系统：表达、使用与模式

第三篇　程序源流：从计算到系统（下）

第七章　应用开发基础

第一节　应用开发的背景与成因

第二节　应用开发技术

第三节　开发视角下的工程问题

第四节　应用程序设计语言的复杂性

第八章　系统的基础部件

第一节　分布

第二节　依赖

第三节　消息

第四节　系统

第九章　系统的基本组织方法与原理

第一节　行为的组织及其抽象

第二节　领域间的组织

附录一　“主要编程范式”及其语言特性关系

附录二　继承与混合，略谈系统的构建方式

附录三　像大师们一样思考——从“UML 何时死掉”谈起

附录四　VCL 已死，RAD 已死

第四篇　架构的思想与指导原则

引言　架构师的思维

第十章　你所关注的系统

第一节　了解系统的过程

第二节　知识的构建

第三节　最初的事实

第十一章　架构是过程，而非结果

第一节　架构师的能力结构

第二节　系统架构与决策

第三节　架构的表达与逻辑

第十二章　架构原则，技艺、艺术与美

第一节　架构原则

第二节　技艺、艺术与美

附录一　做人、做事，做架构师——架构师能力模型解析

附录二　专访：谈企业软件架构设计（节选）

参考资料

图表索引

总论

领域角色的关注

第一节　什么是领域角色的关注

屁股决定脑袋，这其实并没有什么不对。但首先不要关注屁股的大小，而要关注它的位置。在一个系统中，如果你都不知道自己坐在哪里，那么绝对不会知道自己该说什么、该做什么。位置的价值并不存在高下，也不可以拿某种方式去度量。尽管我们可以用收入或财产去衡量

一个清洁工与一个老总的身价差异，但你无法去衡量“清洁一间厕所”与“看一个小时报告”这样两件日常事务之间的不同意义。清洁工与老总，这两个角色所关注的对象本身并不同质，因此无法将这些具体事务纳入到同一个体系中去评判价值。

关注位置，是观察这个系统的一个视角。

你在哪里？是谁？在做什么？

作为一个系统的组成部分，如果要观察这个系统，那么就必须清楚三个问题：你在哪里？是谁？在做什么？从这个系统中别出了“我”，才能分出“其他”，才能分清“我与其他”之于系统间的种种不同影响，从而把这个系统分析透彻。大多数人在做这样的分析的时候，忘记了观察者是“观察—被观察”系统中的一个组成部分，忘掉了“我”的位置，因而少了一半的观察。

《大道至简》这本书通过“工程层状模型”（EHM），从“实现者”这一角色出发，并论及“团队”和“经营”角色。但是——如同上面的问题一样，EHM 模型将这些角色析别出来的时候，也少了一半的观察。举例来说，“实现者”是程序员，“被实现者”是程序，在《大道至简》中却甚少论及程序的本质。又举例来说，“团队”的组成是项目成员，要真正成为“团队”则还有赖于项目目标，在《大道至简》中也只讨论到成员问题，而少掉了对约束或设定目标的那些角色的思考。至于“经营”角色，也存有相似的问题。

反思“工程层状模型”（EHM）的本质，是在讨论一系列工程相关问题在一个轴向上的延伸变化，以及这个轴向的不同领域间的关系。如图 0-1 所示。

图 0-1　EHM 图及其隐含着的轴向

然而这个轴向带来了一个致命的问题：难于承载新的领域角色的加入。换言之，层状模型中加入新的角色就会带来新的分层与界面（关注点），这暗示着该模型下的世界是一对一的、面对面的。这个问题使得我在 2006 年间陷入了一个困境，不知该如何将“架构师”这个角色放入到这个模型中去。一旦我尝试在 EHM 模型中为架构师加入一个领域或一个层次，EHM 立即就崩溃了——无法再解释清楚。

其实原因就在于，EHM 对问题少掉了一半的观察。

领域角色的关注

在我们对 EHM 模型背后的全景——软件工程——作出本质性思考的时候，“在哪里？是谁？在做什么？”这三个问题的提出，将会带来一个全新的视角。因为这三个问题代表的是：领域，角色，关注。

如此前所讨论的，我们可以将 EHM 模型简单地抽象为一个单向的轴线（对这里要讨论的话题来说，轴上的刻度是没有数值意义的），如图 0-2 所示。

图 0-2　将 EHM 图抽象为单向的轴线

在《大道至简》中我讲述过一个细节，即所谓的“工程中没有 BOSS”。简单地说，就是经营者这个角色与我们要讨论的工程问题“几乎”无限之遥远。所以我们先把这个轴线向右无限延伸，也就是说，首先要在工程问题中析出“经营”这个角色，如图 0-3 所示。

图 0-3　轴线中的经营角色

然而，我们又必须认识到：任何一个有意义的工程实施，都是与企业的最终利益相关的，因此也必然会受到经营决策的影响。这种影响可能会很直接地立刻体现，也可能在相当长的时间之后才会表现出来。因此在上面这个图例中，我用虚线来指明这种影响。这一点在后续的讨论中将会相当重要。

剩下的是两类角色，一类是实现角色，另一类是团队角色。在《大道至简》中我们强调了二者本质的不同，以及我在跨越这一边界的过程中最主要的醒觉：语言只是工具。在这一过程中，我留意到：

若某角色在“实现”与“团队”两个领域边界上切换，其成本消耗是不可控的。

由此，《大道至简》中的 EHM 工程模型通过一个界面（临界点）来分隔二者，认为最好的法子是，让实现角色根据不知道所谓的工程，而又在遵循工程的原则下来推进项目。这种分析和推论有一定的合理性，但又忽略了其背后的本质原因，这一点在后面的讨论中将越来越明显。

在使用“在哪里？是谁？在做什么？”这一工具来仔细分析这两类角色时，我们会发现他们所在的领域也是有区别的：实现角色是在技术领域，团队角色则是在工程领域。技术领域关注的是实现的细节，即通过何种方法能将目标有效地实现出来，因而会追求这一实现过程的最优解；工程领域关注的是团队及其所应对目标的规模，在大多数的情况下，这一角色期望控制这一规模以使“目标、资源与质量”可按某种预期、整体地得到保障。有趣的是，从技术领域来说，一旦更细节的或者更宏大的实现成为可能，那么他们将毫不犹豫地将这种“可能”升级为“必须”，并为之充满激情；而这一切，往往又是以牺牲规模为代价的。

对于这两个角色，以及其不同的关注，可以描述成图 0-4（与上面的图类似，两条轴线之间的角度也是没有数值意义的）。

图 0-4　模型 1：对 EHM 模型进一步抽象所得的新模型

这个模型直接抽象自 EHM 中的实现和团队这两个主要角色，并在一定程度上将二者对立起来。这可以让我们清楚地看到二者的不同，和这种差异的本质原因，亦即是它们所面向领域的思维方式与关注对象的差异(1)。另一方面：

□ 它狭义地将工程问题等同于团队问题；

□ 它限制了实现角色对工程问题的探讨。

因此它更加清晰地刻画了《大道至简》中所论的“工程”内在的关系和冲突。

谁关注方向问题？

接下来我们问一个问题：最初从 EHM 模型中应当继承过来的“经营”角色，现在到哪里去了呢？

如果任由技术和工程这两个方向发展，可以想见的是：二者永远是存有分歧的。唯一能平衡这两种分歧的原则、条件、限定等，都必然是来自经营角色。因为，正是经营角色：

□ 确定和分解了经营需求并细化成各个目标（例如发起某个项目或计划）；

□ 构建了特定的组织（例如部门）来推进它。

注意，这里的“经营角色”指代着另外的一个领域，而非是指某一个具体的个人，例如 BOSS。

“经营角色”所指代的领域关注什么呢？它既不关注细节，也不关注规模，而是关注于目标的达成。更进一步地来说，是整个的目标簇是否能维持原定的经营方向：一个或多个目标的方向在短时间的迟滞或偏离可能都不重要，而整个目标簇，以及由目标簇所指示的整体方向才是经营者所关注的。事实上，工程管理与技术实现，以及背后的“某个项目”仍只是经营者所关注的一隅，因为他们还有更多要关注的内容，这甚至包括了公司的张三是否需要提职，业界的李四刚刚与王五达成的“战略合作”究竟是烟雾还是毒刃，如此等等。

“平衡两种分歧”是我们需要在这里讨论这一角色最基本的理由。因此在本质上，就“经营”这个领域来讲，它所立足的是“组织视角”，而关注的则是一个方向问题。将这个视角放在上面这张图中，我们得到图 0-5。

图 0-5 模型 2：“工程的组织视角”对模型 1 的影响

所以，我们看到了“经营”这个领域对工程的影响既深刻又浅末，既是必要条件又难以形成约束。这其中不仅仅有经营角色本身的精力问题，也涉及能力问题。因为，要确保图 0-5 所展示的“工程活动”的实效性，那么我们还需要注意到一个事实：

所谓“规模”与“细节”，其实只是“目标”在两个领域中的投影。

如图 0-6 所示。

图 0-6　模型 3：目标在模型 2 中的投影关系

也就是说，“工程的组织视角”还暗含着这样的三条推论：

（1）目标在工程和实现上的投影正确并相互匹配时，项目能最佳推进；

（2）目标的设定影响“工程与实现”整体的代价，较小的目标（例如里程碑(2)）是更易实现的，反之亦然；

（3）真实的方向与现阶段的目标通常有相当长的距离，其实现通常是以组织的倍增为代价的。

上面的第 2 条和第 3 条推论其实和我们现实的观察与实践是一致的。反过来说，也可以认为该模型体现了工程的现实状况。

工程的组织视角下的视图原型

既然上面的模型 3 中考虑的是组织视角对工程的影响，亦即整体上是讨论该视角下的工程模型，那么仍将其纵轴称为“工程（领域）”就并不合适了。事实上，这个纵轴所代表的仅仅是原始的 EHM 图中由“团队（及其组织与功能）”所映射的领域，是一个相对狭义了很多的工程概念。因此，在继续讨论之前，需要先修正一下模型 3 的概念以得到图 0-7 所示的一个原型。

图 0-7　模型 4：对模型 3 的概念修正

首先，我们明确了模型阐述的主题。整个模型被称为“组织视角下的工程视图”（请留意这仅仅是该视图的一个“过渡版本”），意在将这张图的整体视为对“工程”的描述。

其次，我们将纵向轴称为“项目管理”。这个领域中的角色围绕一个“明确的目标”的投影工作，主要职责在于管理其规模（scope），包括对团队组织、产品特性、项目质量、消耗成本等进行明确的或可预期的管理。以现实的工程角色为例，可能包括团队负责人、项目经理、产品经理、市场经理等(3)。

最后，我们将横向轴称为“技术实现”。这个领域中的角色围绕“与‘项目管理’角色相同目标”的投影工作，主要职责在于实现其细节（specific）。以现实的工程角色为例，可能包括工程师、设计师、分析师等。

上述“技术实现”与“项目管理”二者所关注投影的原始目标“应当”是同一的。在现实的工程中，我们通常称之为“产品”。(4)

VEO 模型：架构角色出现的必然性

对于一个小型的组织，或一个较短期的目标/方向来说，若要求“经营者”来保证两个投影的原始目标同一，并在实施过程中持续稳定，是有可能做到的。这也是一些小公司或小型团队能有良好的组织与合作的根本原因：经营者直接参与规模与细节的平衡。但是，在以下时候：

□ 规模持续扩大、技术渐趋复杂；

□ 经营者的方向与阶段目标间的距离越来越远；

□ 方向由多个方向簇构成。

经营者将难以通过亲历亲为的形式来实现上述的平衡。这些情况下，经营者通常需要通过组织调整来保证其战略推进的有效性。

“组织”既是一种经营工具，也是一种管理工具。所以无论经营者还是管理者，都有可能使用这一工具带来系统整体或局部的变化。换一个角度，管理者其实也可能参与或直接行使经营职责。因此《大道至简》中所述的：

你可以更直接地观察到“经营者”与“组织者”之间的差异。例如公司的大小股东是“经营者”，董事会通常是解决经营问题的地方；而总经理、执行经理及各个部门经理则是各级的“组织者”，经理办公会则是解决组织问题的地方。

这样来将“组织者”作为角色讨论是并不适当的——“组织”，是管理职能的一种工具而非其全部。所以《大道至简》其实是用一种极端情况来区分出了“经营角色”，使我们在 EHM 中的模型可以被讨论而已。

经营者选择组织工具而非其他，是有一定合理性的。首先，模型 4 在“方向”轴上的缺失是局部问题而非全局问题，因此通过组织工具来调适不会带来全局性的风险。其次，经营者可能并不期望颠覆正在进行中的阶段性目标，因此选择组织工具而非战术手段（例如裁撤项目）相对会更为温和。

经营者需要在模型 4 中解决的问题是规模与细节的平衡，以使得工程角色的实施与目标、方向的设定一致。正是这一需求导致架构角色的引入，因为“架构”角色本质责任与这一需求不谋而合。

所谓架构，包含了“范围”与“联接件”两个方面。“架构”一词源于建筑学，中文所说的架构，意指“间架结构”。其中的间、架，在建筑中是对房屋规模的度量用词；结、构则是指建筑的关键位置上的技术构件。

但是，如果我们将架构角色锁定在“某种描述范围与联接件的文书”这样的产出上，无异于将架构角色当成技术工人：使用某些工具，生产某种产品。我在这里讨论“架构”一词的本义，是强调应从本质特性上来看清架构角色所关注的方面。而这些方面，又与在现今软件工程中经营角色对软件系统的关注，例如问题的识别与控制等等存在一致性。正是因为这种一致性，由架构角色来介入模型 4 中的“方向”这个轴线所指代的领域，才会成为一种组织选择的必然，如图 0-8 所示。

图 0-8　模型 5：组织视角下的工程视图

在从 EHM 模型推进到图 0-8 所示模型的过程中，我们让经营角色介入进来，又渐渐地将这个角色分离出去，代之以“系统架构”这样的领域与角色。以下我们把工程视图模型简称为 VEO 模型（View of Engineering Organization）。在开始后续的讨论之前，我们需要再次强调引入架构角色的本意与背景。

（1）目标。架构角色围绕一个阶段目标，以及该目标在规模与细节上的投影工作。这是能将架构角色与其他两个角色纳入同一个系统（具体的工程实施）来讨论的前提。

（2）方向。架构角色在方向领域上与经营者（或更宽泛地称为架构需求的提出者）保持一致，了解阶段目标与方向之间的关系，并通过架构产出、指导、推进和实施等一系列工作来把握这种关系。

（3）范围与联接件。架构的主要产出是对范围进行的约束，对目标的关键构件之间的联接件的设定。并且还需要在实施过程中调适架构最初的约束与设定，平衡由时间、信息等因素带来的目标与方向之间的衍变。

第二节　基于组织视角的观察

我曾经说“记事本(5)这样的软件不需要架构”，这句话并不对，因为彼时我不能将记事本作为一个系统来看待。事实上“系统”这个词并没有“大小”的限定条件，就如同说“人”这个概念本身没有“群体”的限定条件一样：并不是说一群人是人，一个人就不是人。

从组织的视角来看一个系统各个独立的部分，这些部分都可称为角色。

系统中的不同角色

在（系统）架构角色的眼中，目标是一个系统。所以记事本既然是一个系统，则也可以有工作在该系统上的架构角色。记事本系统与整个桌面操作系统之间仅存在复杂性的区别，这在

VEO 模型上体现得尤其明显（就我们这里要讨论的话题来说，图 0-9 中规模的绝对大小是没有数值意义的）。

图 0-9　模型 6：不同规模的系统在 VEO 模型上的映像

现实的工作中，我们没有为记事本的开发指派一个“系统架构师”的原因是：它的规模与技术复杂度一个人就可以控制(6)。但这并不是说，这个“个人”在软件开发过程中就没有过规模、细节、方向这三个领域上的思考。简单地说，我们在很小规模的软件开发中，可能是由一两个开发人员同时负担了管理、架构、实现三类职责而已。所以——

我们要讨论的是“角色”，而非某个职务或具体的人。

以架构为例，我们讨论的不是“谁来做架构师”的问题，而是“架构这个角色应该起到什么作用”的问题。首先，最迫切的问题是要弄清楚项目目标，这是必然的。在这个目标被作为一个产品指派到某个项目组之前，架构师就应该清楚在更大范围的“系统”中，当前这个目标的真实意图。例如记事本从 Windows 1.0 开始就在系统中存在，有着诸如此类的原始设定：

□　作为对 DOS 等早期命令行系统中的行文本编辑器的替代；

□　是操作系统缺省的文本处理工具。

而在这个目标之下，规模问题是项目管理这个角色所主要关注的，更确切地说：

规模问题的核心是项目目标的投入与产出。

所谓投入，包括时间、人力、资金、设施等，项目管理者必须考虑项目在各个阶段的投入情况并确保它在一个可控的规模。所谓产出，是指项目目标（例如产品）的特性及其细节，项目经理必须保证这些特性是在一定边界上增减的，是可测试与交付的。

这看起来与质量的三要素，以及软件工程的体系层次等模型有隐约的相似，如图 0-10 所示。这些用来阐述软件项目或软件工程的传统模型，是从工程质量保障或实现方法的视角来考察工程的。然而关注质量平衡或关注实现层次，仅是在规模控制中的手法，是部分的要件而非其全部，例如在《大道至简》第三版中，就将做得“更多”或“更好”等等都作为规模失控的一种形式来看待。

图 0-10　软件工程的质量三要素和体系层次

所以事实上架构角色与项目管理角色都在关注记事本的规模问题，但仍然存在着一些不同。例如，如果相较于复杂的 jEdit、Editplus，或者便捷一些的 Notepad++、Win32pad 等来说，有人提出了类似“设置字体颜色与背景颜色”这样的特性时，架构角色可能会首先考虑以下

因素。

（1）记事本作为操作系统的默认组件，其外观表现和交互特性应当是由系统的全局设置来控制的。对于前者，例如桌面主题导致的记事本前景与背景变化；对于后者，例如系统默认打印设备的设置，或者输入法设置。

（2）如果操作系统的默认设置不能影响到记事本，则系统的其他默认组件也会存在类似的问题或需求，这意味着整体实施的复杂度会增加。

（3）类似于 jEdit、EditPlus 等产品的用户只是操作系统用户群体的一个较小子集，其需求不具有代表性。

（4）以记事本通常处理的文本长度来说，是不需要用颜色来区分文本的不同部分的。

对于项目管理角色来说，他否决这项需求的理由会更简单直接：

（1）在当前的记事本版本中，未定义该项特性；

（2）该项特性与原始的设定“文本处理工具”没有必然关系，可以延后决定；

（3）该项特性在来自产品、市场等各方的报告中有明显分歧，存在实施风险；

（4）在项目实施阶段，增加该特性对项目过程控制存在不确定的影响。

然而与上述两类思考不同，技术实现角色将更多地关注实现的细节问题。其中一类问题是与项目经理共同关注的，通常与项目背景以及实施环境有关系。例如：

（1）可能的代码量与要求的代码质量；

（2）后续维护人员的水平以及因此所需要的注释详细程度；

（3）开发环境与测试环境的部署以及性能。

当然，类似于在何种团队中工作，以及开发部门中是否可以玩桌游等，也是技术实现角色经常关注的问题。不过这类问题的特点是：与具体项目并没有关系，因此大多数情况下会由公司的技术部门或整体组织来平衡(7)。

在具体的项目中，开发人员通常更关注的是另一类更加细节的专业问题，例如：

（1）产品性能的具体要求，例如运行在何种设备上，以及定量的稳定性要求如何；

（2）采用的语言、框架、程序库，其技术复杂度如何，以及资料是否翔实等；

（3）待处理的标准文本格式规范，例如 UTF-8 编码以及 BOM 头规格；

（4）需测试操作系统初始环境中所有类型的文本，包括.ini、.xml 和.reg 等；

（5）该产品应由多行（ES_MULTILINE）的 Edit 来实现。

这些问题的部分或全部也会对项目的规模构成影响，从而改变项目原始目标的设定。例如说，如果记事本不是由标准的 Edit 来实现，而是使用 RichEdit 来实现，那么它就具有了与写字板(8)相同的规模与特性。所以架构师同时也需要站在技术实现的角度上，考虑技术的选型问题，因为只有架构师知道“系统的其他部分还存在一个使用了 RichEdit 组件的写字板产品”，并且又具有控制记事本向写字板演化的职责。

我们看到：

□ 架构角色不单单关注记事本自身的规模，还关注其外在系统的规模，以及二者的关系，例如他关注记事本与写字板之间同质问题，并设定原则来辨识它们；

□ 技术实现角色则关注实现技术是否便捷、有效，以及是否能把控实现过程，他的这些需求来自于对项目产出的责任，以及对自身实现能力的衡量；

□ 而对于项目管理角色来说，一旦产品规格书上有确定描述，他就不需要关注“该项目是不是把记事本做成了写字板”。

当然，架构师就要为类似的规模失控问题挨板子——这也可能是产品架构师的问题。不过要到本书的最后一部分，我们才会来讨论架构角色的分类问题。

透视：一体的两面与多面

回顾 VEO 这个模型与项目的关系，我们可以将“整个系统”所涉的子系统划分在不同的业务领域中。这时我们会发现，VEO 也展示了子系统与系统全局之间在方向上的不一致性。它们既有可能是同向或基本同向的，也有可能是异向的，或者无关的。图 0-11 部分展示了在 Windows 操作系统中的“多媒体子系统”可能涉及的一些细分子系统、业务领域。

图 0-11　模型 7：系统局部与全局的关系

在将操作系统或“多媒体子系统”的细节投影到 VEO 模型时，我们会发现：

（1）目标跟“规模和细节”之间是一体两面的关系；

（2）系统全局的目标，与“不同子项目的规模与细节”又是一体多面的关系；

（3）局部目标与全局目标并不存在简单累加关系，因此全局规模与局部规模也不存在累加关系，“细节”轴线也存在相同的问题；

（4）目标在不同方向上越分散，子系统在规模与细节上冲突的可能性就越高，系统复杂度（管理成本与实现成本）也越高；

（5）不同的方向间产生的内耗极大地增加了系统的代价，而规模与细节的失控只是这个问题在两个轴向上的表现。

面对这样复杂的系统分析，架构角色应当要有能力来回顾（review）各个子项目，有意识地放弃掉一些不重要的、投入与产出关系不明朗的，或者对系统全局会有负面影响的子系统。同样，架构角色也可以将部分力量聚焦在一些子项目中，以使战略方向更为明确和落到实处。最后，也是最重要的，架构师要能把握全局力量的投放，对于某些有远见的方向，或暂时不清晰的方向予以持久的关注，这是架构师在系统整体调控能力上的最终体现。

组织：组织力下的 VEO 基本模型

VEO 模型表达了技术、管理之与架构，是一体的两面，这种关系可简单描述为图 0-12，但这只是一种理想状况——准确地说，是相对理想的状况当中的一种。

图 0-12　技术、管理之于架构是一体两面的关系

在我们最初做软件开发工作的时候，例如个体开发项目中，所谓技术角色与管理角色是一体的，这是最简洁的开发模式。接下来，当项目规模大到一定程度时，我们发现需要一个管理角色来参与项目的过程，让他通过控制这一过程来限制项目的规模，保障输出质量与投入成本的平衡。这催生了传统的软件工程模型。

传统软件工程模型其实就是一个软件开发方法论在过程与工具上的投影。

然而尽管我们已经基于这样的模式实践了几十年，却仍然在不停地探索管理与技术之间协调工作的方法。从这个角度上，即从实施的视角，对传统的软件工程模型加以审视，如图 0-13 所示。

图 0-13　基于实施视角，对传统工程模型的另一种抽象描述

这事实上也说明了传统的软件工程为什么都是在某种方法论上的具体实施。然而我们观察实施视角下的软件工程，（基于某种方法论，）它除了引入更复杂的过程/流程，以及更多门类的工具来协助工程的推进之外，并不能解决组织问题。

现在，为了保障更大规模下的目标，我们引入了架构角色。架构这一角色，在组织关系上介于管理与技术二者之间，在工作内容上与二者有所重叠，在职责权力上又难以对当前项目与产品产生直接效果。这三个方面的关系，使架构角色在组织结构上显得尴尬。而与这种尴尬

的状况不匹配的，是它可能带来的负面效应。

架构角色对目标投影的失控将不可避免地导致组织规模扩大，以及管理与实施之间失联，图 0-14 表现了这样一种状态（图中的角度值仅用来与 VEO 基本模型比较，并非有确切含义的数值）。

图 0-14　模型 8：组织结构调整带来的问题

所谓组织失联，是指在组织结点之间（例如管理与技术）所工作的方向不一致、沟通成本增加，以及信息不对称——例如架构角色所展示的“目标映射”在其他角色有不同理解。进一步的结果，随着失联愈趋明显，最终有效的工作集变得越来越小。与此同时，组织的规模却在增长，管理边界与成本也增大了。这一过程，其实就是所谓的组织结构调整所带来的内耗。

积极的管理或技术角色会趋向于弥补这种内耗，无论是二者谁占据主动（话语权或说服力），其结果无非是图 0-15 所示的两种情况中的一种。尽管这两种模型从本质上来说是一样的——因为“实际的规模”中的内耗与模型 8 中是等量的，但是架构失控偏向于某一侧，其结果却可能不同。

图 0-15　模型 9：积极的组织适应带来的效果

在模型 9a 中，由于架构角色对细节失控，因此他只能尽量争取管理角色的支持。如果他有足够的组织责权，管理角色又能很好地配合，那么业务会朝向架构目标推进。但即使如此，由于技术角色所能感受到的方向是趋于变化，所以技术调整的成本会相当高（例如技术人员离职、框架废弃等）。另一方面，如果架构与管理无法配合工作，那么最终的结果是管理能力决定了项目是否“按时完成”，但项目产出与原始需求会相距甚远。

在模型 9b 中，由于架构角色对管理失控，因此他只能尽量争取技术角色的支持。如果他有足够的技术影响力，并且可以在需求和目标上与技术角色达成一致，那么业务也会朝向于架构目标推进。与上一个例子类似，管理角色将会缺席，事实的情况变成了由架构角色来驱动一个技术产品的研发过程。架构角色可能会在细节上与技术角色沟通，而忽略了在时间、资源等成本上做好控制。更或者存在另一种可能，即架构与技术角色仍然无法达成一致，那么最终的“业务推进方向”将变成“技术创新产品”——当心，完全无规划的技术创新，既可能是全新的业务，也可能是耗尽资源的垃圾。

面对组织失联，积极的愚公可能挖山不止而终无所成（或神话般地感天动地），消极的管理或技术角色则可能会让经营者看到一个“貌似可喜”的局面。当管理、技术与架构角色之间都不能相互配合时，他们各自为阵进而达到一种平衡势态，如图 0-16 所示。

图 0-16　模型 10：消极的组织对抗带来的平衡态势

之所以说这是“貌似可喜”的，是因为这个模型表面来看不存在问题：组织的要义在于能以一种稳定的模型持续推进，而不稳定的组织结构意味着自我调适带来的内耗。同时，这种平衡态势也是组织力影响 VEO 模型的正常结果。如上所述，无论以何种结果（例如大量内耗）为代价，组织的自我调控都将是趋向于衡态的。具体来说，在信息的分享上，完全无分享是最确定的衡态；在目标方向上，各向的牵制是最确定的衡态；在组织规模上，各角色等量发展是最确定的衡态，等等。

所以，如果仅仅构建（或自然形成）VEO 模型中所述的架构、管理与技术角色关系，并依赖他们完全自发地调适，模型 10 将会是最有可能的结果，也将是消耗最大的一种衡态：系统的规模最大而有效的工作集最小，并且在目标和实施上存在不对等和不确定性。

合作：VEO 模型工程的人为因素

我们讨论到了组织衡态的一种形式，即因为“架构角色对目标投影的失控”而导致各个角色的方向/目标不一致，由此各自发展而形成彼此独立的衡态。事实上我们大多数时候处在组织演进的过程中，“不是东风压倒西风，就是西风压倒东风”，因而上述衡态是一个理想的（但并不是良好的）结果。

我们必须讨论事物的另一面，即如果目标投影能准确地映射到规模和细节上，架构角色在这一过程完全履行了自己的职责的话，又当如何呢？如果这一切发生的话，则管理角色与技术角色将完全理解并忠实执行架构角色所述的目标，这一过程在 VEO 图上可以表现为图 0-17。

图 0-17　模型 11：架构角色的职责可能对组织产生的影响

由于目标趋向一致，因而三个角色趋向于合作。随着组织规模渐趋紧凑，组织变得愈加高效，目标细化而又锁定在一定的范围之内，其最终的实施结果无限趋近于目标的预设。也就是说，最终的衡态也可能是三者在关注的方向和内容上相同，仅存在责权范围的不同，如图 0-18 所示。

图 0-18　模型 12：面向合作的组织适应带来的效果

显然，这种衡态是一个理想的（也是良好的）结果。而这种结果——在这一个阶段中——是由架构职责的忠实履行，并与其他两种角色切实合作带来的。这意味着两件事：

（1）架构角色完全理解自己的职责；

（2）管理与技术角色完全理解架构角色的作为，并努力配合之。

然而我们面临的现实并不乐观。一方面，争取管理与技术角色的合作必然要求架构角色在这个组织中存在一定的领导力。领导力并不是组织能力，否则当某人被任命为架构师时，他就应当有领导力了；领导力也不是管理能力，否则找个高管来当架构师就好了。所以，

架构角色的领导力（以及其话语权）另有来处，而组织行为与管理行为只是保障这个目标的两个可选工具。

另一方面，对于架构角色，其责权的重要性与职务的确定性存在明显的不匹配：到底架构是什么、如何做，它的产出以及影响工程的具体方式等，难以在项目团队中界定清楚。无法确定这些行为，也就无法领导这一个阶段的工作，所以当“角色的关注”变成不可实施的空话时，领导力本身便没有了价值。这从另一个角度说明，架构角色的领导力，部分来自于实施架构过程中的确定行为。

是所谓行胜于言，清谈无架构。

调适：变化中的 VEO 模型

有三种组织模型能够描述“有着某种领导风格”的组织，分别是 Owner、Center 与 Core。在这三种模型中，领导者将自己作为一个点，代表、凝聚了这个组织，或作为组织的一个缩影。但这样的组织模型只表明“领导力”的一个结果，即“如果某些角色是有领导力的”，则他们的推进结果可能使组织演变为这样的一种形态。我们不能反过来说，一个结果（如形态）就是领导力本身。

VEO 模型在角色职能出现了架构、管理与技术三个方向，从团队/项目组织职能上来说，这三个方向也就存在三个（可能的）领导角色，即架构师、项目经理、技术经理。如 VEO 模型所阐明的事实一样，图 0-19 表明这三类领导角色在组织上是趋于衡态的(9)（领导力下的衡态，以下简称 LoE，即 Leadership of Equilibrium）。

图 0-19　（与领域相关的）领导力下的衡态

如何让 LoE 模型趋向于一个“有领导力”的组织模型，是这一系统中的各角色间存在冲突的主要原因——换个简单的说法，就是大家都想当领导。在另一方面，这些角色将受到更外层的角色的影响，外层角色级别可能更高，或更为具体（如开发团队、产品）或宏观（如更高级别架构决策者、产品线），如图 0-20 所示。

图 0-20　外层角色的影响

许多情况下，这些外部力量左右或影响了具体角色的实施（并不一定是决策）。这个问题随着领导责权属性的不断增加，例如可能出现的跨角色授权或领导小团体的形成，将不断扩大到整个的、全局的系统，如图 0-21 所示。

图 0-21　更多的组织角色构成的阶层

为了解决这一问题——或为了利用这一形势带来的利益，我们通常在某一方面或某一方向上合并角色，如图 0-22 所示。

图 0-22　通过合并来打破阶层（示例 1）：决策者的思想

这样的组织形态一方面可能因为在相应方向上的领导缺位导致，另一方面，也是因为组织中既已形成阶层结构——并由此带来权责分配，因此“自然地转变为（而未必适宜）”由这些角色通过他们的领导力来影响项目。甚至有些时候，你看到 CEO 都被合并到这个角色中来。由于这种合并，项目常常变得非常难于控制，因为角色对项目的影响会因为“管理层次”，而非“项目自身需求”而变化。

然而这一趋势可能会进一步激化。因为，在各个方向的合并在一定程度上是有益处的：首先，它满足了某些组织角色的权力诉求；其次，它确实在很大程度上减少了沟通成本；其三，它更利于构建“那些还不存在的”团队或组织结点；其四，它的失误成本更小，例如对于该组织来说，失败时责任划分更简单，且可以更快捷地改变；最后，它事实上也提供了领导角色短期实践的空间。然而更大范围的合并，通常责权更为集中，而角色与其职能也就更难区别，而且一些时候也不可避免地出现“为了形成对立”而结成的联盟。

图 0-23 所示的模型事实上形成了技术与产品（包括其他一些非技术角色）的对立，在这样的模型下，技术是否“生产”符合需求的产品，是否在做符合公司利益的事等，这些问题都是由被合并后团队的决策者（或事实领导者）来判断的——但这是否可行，是组织权力上的具体考量。图 0-24 所示的模型貌似风险更大，例如它看起来在做“CEO、CTO 与产品线都不知道的某种东西”，但事实上它更经常地出现在创新团队中。

图 0-23　通过合并来打破阶层（示例 2）：实现者的思想

图 0-24　通过合并来打破阶层（示例 3）：尝试者的思想

这也意味着如果“组织形态的合并”在规模与责权上是可控的，那么它既可能产生新的产品/项目，也可能产生新的组织结点。但如果反之，在这两方面失控——例如在不正确的角色上做出了不正确的决定，那么它的影响面也就更大。然而事实上，我们在项目中的思考常常并不那么清醒。例如，会出现某种空中楼阁式的、忽略了开发技术实施可能达到的能力的合并，如图 0-25 所示。

图 0-25　通过合并来打破阶层（示例 4）：规划者的思想

基于 LoE 模型，事实上我们是在讨论“组织的形态”对于 VEO 模型可能发生的影响。这些影响既有正面的，例如权力的扩大、沟通的简化，以及领导力的形成等，也可能有负面的，例如联盟与对立的形成(10)。回到最根本的问题点上来说，真正形成对立的，是领导力与组织力在 VEO 中的价值。从这个根本出发点来看，我们会意识到：组织力与领导力在本质上是存在一种“堵与疏”的关系(11)，如图 0-26 所示。其中：

图 0-26　组织力与领导力对组织形态的不同影响

（1）组织的基本形态是各行其事，因而组织力的根本在于构成确定组织结点的权责，并进而明确系统规模与代价；

（2）领导角色的基本职能定位在于形成系统的合力，因而领导力的根本在于消除内耗。

对于 VEO 模型来说，驱动其变化并趋向良性的是领导力，而其依赖的事实基础却是受组织力影响的 LoE。这是两个矛盾又统一的结构，也是大多数结构的内在形式。对这样的结构加以调适，其关键在于使用者的思想、方法与眼光。

第三节　工程的本质问题是组织

工程不是“做”的，是“组织”的。这个“组织”，既有名词的含义，也有动词的含义。除了这个层面上的意思，它还意味着一旦没有确定的组织模式，那么一个具体的工程也就难以落足——即使是个体工程，也有一个确定的组织模式。

组织的源起及其发展的过程，是一个核心问题。

组织的背景

然而在实践中，并没有人注意到工程的组织问题，而只看到了工程实施的结果。这些结果包括 RUP、UML、XP 以及未知的种种模型、框架和方法论。所有这些看起来可以一统天下的，或致力于一统天下的方法，都最终走出了它们原本栖身的岩洞，走入莽莽苍苍的草原或森林；而它们还在认为可以通过敲击岩壁去听发出的声音，并以之判断树洞、沙穴或草坑的居住年限。

它们的共同问题是，没有考虑到具体方法实施时的组织背景。

组织是一个项目的原始背景，组织的构成是人，是人群。工程中最终要解决的，不是具体方法的问题，而是将不同性格、性质的人组织成群体并实质推动的问题。从这个角度上来说，《集体行动的逻辑》所带来的思考远比《人件》、《人月神话》来得更加深刻和直接。

无论是以人为工件，还是以“人/月”为度量衡，根本上还是把人作为个体来思考的。虽然这些直指个体的思考是必要的，但也无助于组织与该组织下的集体活动。

契约社会与人情关系之间的区别

中国的传统社会模型是基于人情关系的，这与儒家的社会阶层化有着密不可分的关系。儒家从根本上来确定了天地人伦的关系之后，人们的社会行为、族群关系以及家庭婚姻等都是在这一背景下的自然发展。因此，我们讨论的同学、朋友、党派、宗族以及更简单的门当户对等，都是人情关系的基础单元。并且这样的基础单元深深地植根于我们的民族文化中，是文化内涵的一部分，也是构成我们民族化的人性的一部分。

人情作为社会成本的一部分（如果将规模缩小到一个组织，则是组织成本的一部分），是我们这个社会的现实状况。这个成本在不同的领域与不同的事务之间，并不是均匀分布的。有些领域例如产品生产，人情成本会少一些，而另一些领域，例如组织构建，则会多一些。粗略地分析这一分布的模式，大体上（并且与后面我们要讨论的契约社会比较来看）可以认为：所需要决策的事物对人与人之间的信任的依赖程度，决定了人情关系的比重。

我们将人情社会中的人情成本转嫁为信任成本的缘由在于，很大程度上来说，人情是构成社会信任（以及组织信任）的要素。对于任何一个事物的决策，信任所带来的收益往往是别的努力所无法达到的，例如开上 100 小时的会议抵不上老总的一句口头传达。这看起来是政治与权力在起作用，事实上也可以看着对权威的普遍信任。

我们讨论的“组织”，既是社会背景下的一个局部群体，也是社会行为模式下的一个普遍模式。简单地说，我们的大多数组织都是人情组织，而这样的组织下也都是人情成本转嫁为信任成本的。正如某天，一个开发人员对我说他想转职去做管理，我问他的第一个问题是：你认为去做管理，有多少人会听你的？同样的问题是：你去做产品，有多少人听你的”，或“你要去创业，又能拉上几个兄弟的人马？

当这样的问题提出来时，我们的大多数开发人员都哑口无言。因为多数工匠思维下的工程师是关注做事，而不关注人情的。在我们的社会与组织背景下，事做得好不好，只是整件事情中的一个比重较大，但不十分关键的因素。同样地，作为一种“集体行动”的组织方法，与“在做的事情”关系也并不大。例如我常常让别人思考：“为什么外行可以当领导？”我的意思是说：领导作为一门技术，显然也是有其方法的。所以，反过来也可以问：一个“很有领导能力”的领导，为什么不能做（这件事的）领导？然而，这样的逻辑——至少在我们的背景下——掩盖了基于人情与信任的那些逻辑，即我们的组织基础以及由此进化而来的行事逻辑。

然而我并不是要将这样的组织与逻辑引入到我们的“工程”中来。正好与此相反，我是要将这些因素从工程中排除出去。我是想彻底地问清楚：工程到底是什么呢？在我们的讨论语境下，所谓工程，指的是一种行事方法。如果我们不能清醒地认识到“方法是依赖背景的”，则我们不会转而去看一看这些背景，也就不会把这些背景摒弃在我们的工程之外，也不会善用之，使其变成有益于工程方法的一部分。然而这仍然存有巨大的风险：工程究其底里仍旧是事，它必然有着方法的背景。那么如果我们把“这样的组织与逻辑”推出去了，又能把什么样的东西拿进来呢？

我的答案并不是“契约”。

只有基于契约的利益得失严重影响到社会成本时，“契约”才能够作为组织手段实施。这是基于“自利”这一思维模式的、必然的、必须的组织结论。换个简单的说法，就是如果毁约代价大，则大家都守约；否则，大家该人情还人情，该无耻还无耻。就如同没有极端代价的、公义的法律约束，大家就会犯法一样——这看起来很是好笑，但我们的社会在底子里就存在着这样的一个模式(12)。

有没有可能换一个视角来看这个问题呢？比如信仰，又比如文明。但如果没有与法律这种“人与社会的契约”类似的东西的话，又何以判断行为是否“文明”，或是否是“信仰所确定的一部分”呢？看起来，“道德”可能是一个很好的解，也就是“讲道德”大概是文明的，或符合某种信仰的规义的。但“道德”又是什么呢？

我一直说我们的思维模式是理性而又逻辑的，因此我们必须先说清楚“是什么”。然而在我们的实践中，是不是要这样才没有“错”呢？看起来，我们一方面在花极大的时间去讨论这样的“正确与否的问题”与“问题的正确与否”，却在一些现实中要去解决的“事”上寸步不前。

换言之，我们是不是可以把大家普遍看起来“道德一些”的东西拿来就用，而暂且无视它学术的、逻辑的或严谨的定义呢？

回到契约的本质上来。即使我们承认一个人的选择是自利的，承认这个选择的背景是阶层化而非自主平等的(13)，但为什么不能将选择的结果视为契约的呢？“诺”于一职，“诺”于一事，于“人情社会”真的就是挑战么？我认为不是的。所以在我看来，工程在具体上于我们的解，便在于大家对事的负责与不负责，而不在于对组织“是否契约”的讨论。若变了法子要去改变组织，或要去理解组织的全体，既无益于我们的一个具体的工程，亦无益于我们自己的选择——如果是后者，那你的选择应该不是“做一事”，而是组织的革命(14)。

对此，承认还是漠视？

学术的讨论通常会将契约与人情二元对立起来，试图非黑即白地讨论这二者。大多数情况下，一个论者必须表明他对两个观点的态度，并且义正辞严地批判另一种。出于学术的讨论与真相的挖掘，这样极端的思维——在一时之间——是必要的。因为若不如此，我们便难于看到一面或另一面的真相。有了这种学术讨论的氛围，若有人说“契约社会不错，人情社会也很好”，那么就会被斥为和稀泥，变成两种观点同时讨伐的对象。大家争来讨去，把第三者打倒了，又开始双方的互殴，这是市井间打架的惯常路数。

改革家也会跳出来。他们往往关注旧事物的败亡，以及新事物全面占领阵地的那一刻。这种情况下的局面通常是：旧的东西被没有了，即使有一点点可能存在的痕迹也要用黑布挡了起来；新的东西看来已经存在了，至少每个人的口号中都这样地呼喊着。然而这时，这些“改革家”（例如那些来向你宣传的布道者们）立即就消失了踪迹，因为他们也在瞬间失去了主意，不知道应该怎么办才好了。——他们的义务与职责，原本只是改革，无对象可革了，这样的角色也就不复存了。

所以真正有意义的改革是革新，而不是革命。革新是容旧渐新的，革命是推翻了事的。然而既然如此，我们便会推知一个重要的事实：旧的，必须与新的共存着非常长的时间。这意味着，这一过程中旧的必须渐渐“懂得”新的，而新的也必须渐渐“容得”旧的。于是旧的不再旧，而新的也必然不再是最初设定的那个“新”。因为，“新”的自身也必然是在发展着的，如我一再说的，即使那些“（看起来更正确的）新东西”是真理，也只是一时暂存的“真理的某个表象”，我们必是在追求真理的过程中揭示它的各个方面，而最终——以人类的终极来说，得到一个所谓的真理的。

看起来我是把这个问题变成了一个哲学的讨论。但问题在于，对于“承认还是漠视”这个设问来说，观点无非是“承认”、“漠视”以及“一边承认着，又一边漠视”。这样的观点在二元论的、一元论的思维体系中，总是要去打倒一个两个的。于是我们大家就变成了一群聒噪的村妇，为着地头上的几株桑树占了谁家的地盘而厮打不休。

然而又过了些年，无论吴、楚都归作了秦，这些厮打也就全然失去了价值。

学术与践行的区别就在于此。能用就用是践行的第一原则，小平同志的话更简单直接：不管黑猫白猫，能抓老鼠的就是好猫。而在我们的工程实践中，我们的背景是人情关系，是阶层管理，是不完善的规章流程……我们若是无视这些而去实施某种“标准化的”工程，那是有革命的胆量却没有改革的眼光，勇而无能；反过来讲，若是受了这些的牵绊，便又陷入了人事的、章程的困局，因循守旧故而难有破局。

然而我的问题一直不在于这样的一个“看似合理而又毫无意义”的结论。亦即是说，偏左亦死偏右亦亡，是通常学术所谓“骑墙派”的观点。——这些骑墙看戏的，通常置身事外，将看戏视作一种乐趣亦或是一种必须。尽管这样的“结论”无助于任何的实践，然而这一“过程”——

“停下来，看看你们在干什么，在什么样的背景下干”。

却是实践所必需的。

所以我认为，工程的问题，尤其是一个具体工程的问题，还是要回到工程的本体——这个组织与系统的背景上，重新地审视才对。无论那些争吵不休的学术家，或是那些埋头苦干的践行者，都应该仔细看看我们的工程环境，看看那些“正在做”的做法，看看那些在组织中被记着“人头数”的工程师，以及被记着“大头数”的管理者们(15)。

我想若是停了下来，哪怕一刻，也是有益于我们前行的。

组织与系统

如果“你”已经是组织角色中的一员——例如架构师、项目经理或程序员，那么当跳到系统之外去思考时，你会提出的有关组织的问题是：那个人是谁、在哪里以及在做什么？人总是能看清别人而无法认识自己，因此当你看到“那个人”其实就是“你”的时候，你就已经看到了整个体系、整个体系中的每一个人与每一个“我”——当看到“你”的时候，你已经游离于自我之外。

于是，你的问题是不是在另一个体系上，或在你当前的体系上，只是你的视角问题。例如，当你辨别出了所谓的“另一个体系”以及“当前的体系”，那么只是你因为观察的时候站在了两个体系之间或者前后。如果，你能够站在由二者构成的“整个的体系”之上，那么你的问题就被投放到了这“整个的体系”之中，变成了一个确定的问题。

而所谓的“整个的体系”，就是整体。

你看得到“整体”吗？

所谓“架构师要有大局观”，其实就是指架构师对系统全局的思维能力。例如，当你把需求方纳入你的思考范畴之后，你就来到了更为复杂的工程背景设定之中。这时你需要考虑的问题，将不单单包括一个项目的具体实施过程，也包括一个项目如何产生与交付。影响产生或交付一个项目的因素极其复杂，从社会的产能，到各国首脑的多方会谈，甚至到跟你握手的那个客户经理能否在楼下顺利地找到停车位……这所有的一切，构成了一个项目过程的外部因素。而你的组织，只是飘摇在这些外因之下的、微不足道的一把木片而已。

只有站在组织之外，才能看到组织的整体。无论那把木片中的某一块（例如架构师这个角色）的材质是何等的优良，也不能决定自己以何种形式、能否足够优美地飘摇。因为局部不能决定系统，它只能影响后者，或受后者影响。最简单的实例是，你的项目过程再完美，但老板说客户已经破产了，于是你的项目就终结了。在这个例子中，“公司+客户”之间的关系，决定了项目的成败，与团队、组织、过程这些因素全然无关。

无论如何，组织问题以及工程实施问题仍是我们讨论的“系统”的具体视角，并不是其全部。但我们不可能要求一个或某几个角色去关注所有的系统问题。本书也不是“系统论”，因此并不以讨论系统的整体、关联、等级、平衡、时序等这类问题为主旨。本书希望通过“领域角色的关注”这样的话题，来发现各角色所在领域中最应当关注，以及对系统影响最大的问题集。因此，本书的后续部分将讨论这些问题集以及不同角色的思考，至于这些思考过程与问题的解决方案是否是某个工程方法的实施要件，是一个应用话题，而非本书的焦点。

————————————————————————

(1) 思维方式并不是思维对象，前者是基于后者而渐渐形成的思想方法；思维对象也并不是关注对象，（在同一个领域下）前者是后者所带来的、在意识中的映像或抽象。

(2) 里程碑（Milestone）是“靠改进特性（Feature）与固定资源（Resource）来激发创造力”这一微软的软件工程观念中的基本概念。

(3) 这并不是说项目经理要“管辖或替代”产品经理的职责，而是说在“范围”这个领域中，事实上（在现实中）是并存着这些角色的。“项目管理”作为一个现实职务时，它管理的具体内容是与组织的授权有关的，这在后面的内容中将会讨论到（例如，“调适：变化中的 VEO 模型”）。

(4) 对于“方向”这个轴线上的“目标”来说，项目所管理与实现的，是阶段目标下的“阶段性产品”。同样，以纯粹的“产品视角”来说，“方向”轴线上的目标/产品系列，即是“产品线”。更进一步，经营角色同时关注的是多个产品线上的方向问题，于是一系列产品线所构成的某一个方向上的“方向簇”，通常可以作为一个“战略”的战术实施。在本书中，并不对模型 4 中所隐含的“产品视角”作深入讨论。

(5) 在这里以及后文的讨论中，我是特指 Windows 中的 notepad.exe 这个记事本工具软件。

(6) 我并不清楚在 Windows 开发团队中，记事本软件是不是由“一个人”来开发的。但大多数人认为类似这样规模的产品，是可以由一个人完成的。

(7) 不过对于一个时间跨度很大，或者会持续多个阶段的项目来说，这些问题可能就落到了项目经理的头上，因为他也担负着团队建设的责任。

(8) 在这里以及后文的讨论中，我是特指 Windows 中的 write.exe 这个写字板工具软件。

(9) VEO 与 LoE 描述的是相同模型的两个方面。二者的区别在于，前者面向领域间的差异，后者体现项目/系统整体下的角色冲突。基于系统整体观察时，若从组织形态上来说，应强调 LoE 中的冲突，因为这是组织外在的、形的东西；又若从组织结构上来说，则应强调 VEO 模型反映的领域与角色，因为这是它内在的、质的东西。

(10) 一定程度上，可以参考奥尔森有关利益与利益联盟的形成的理论。

(11) 这种关系不是衡定的而是变化的。例如组织力通常扮演堵的角色，但当系统趋向呆板、滞化时，又通常是用组织力来实现疏的效果。

(12) 至于这一模式诸多的内在与外在的驱动力，还可以有相当多的讨论，但并不是我们这里的话题。

(13) 《社会契约论》是从自由、平等开始讨论的，可见契约本身的成本即是社会关系的成本。基于此，维护契约及其平等性才有必要，因为这就是社会信任的全部出处。然而，如果我们以“既存的人情”与“基于人情的组织”来讨论这一问题，那么人情关系对信任的影响就相当大了，而“维护契约及其平等性”的必要性则会相应地变小。

(14) 做好一职，做好一事，是我对事的求解，而非对组织的求解，亦非对软件工程的求解

（它可以视为从组织视角出发，在具体工程下的一个可能的解）。需要明确的是，所谓“一诺”是基于道德的，而非基于契约的，只是在形式上用到了契约的封皮罢了。

(15) 在大多数的公司中，一个部门或团队只是“你有多少人”，以及有多少个“高管”或“大V、大P”这样的概念。组织管理者对“人/角色的性质”的、不正常的忽视，确是由来已久。

引言

管理中的逻辑

“不太逻辑的思维”，是不是我们用以逃避问题的一贯策略？

不同角色对人的思维方式的要求是不一样的，例如你不能要求音乐家如同数学家一样地去思维——这当然存在特例，但追问与研究它通常是科学家的事情。对于大多数人来说，意识到“自己可能不擅长从事某类工作”，是相当令人沮丧的。

没有人承认自己是笨蛋，于是“思维缺乏逻辑”便成了一种逃避软件开发（或其他某些类型）角色的一种托辞。然而就本质上来讲，逻辑思维事实上是每个正常人的基本能力，例如《大道至简》中谈到过的起床穿衣刷牙等这类日常行为中就包含了大量的逻辑与思维活动。

但是真正深入地讨论“逻辑”却远非易事。西式逻辑学在最早传入中国时，被译作“西方名学”，这是从古典的“名实学”继承过来的一种说法。但是“名实”并不能完整地表达这一学科的内涵，于是在后来的翻译中便渐渐地采用了 Logic 一词的音译，亦即是“逻辑”。所以，逻辑二字，在中文的文字学上其实是无解的。反观“名实学”，它其实只是部分地表达了逻辑的涵义：所谓名，就是一个物的名字或指称；所谓实，就是一个物的本身。

在逻辑学的概念中，“桌子”既能指代现在你眼前的办公桌，也能指代我家的餐桌，那么当我们都用“桌子”这个词的时候，究竟是在“谈什么”就成了大问题。“名正则言顺”讲的就是这么个道理，亦即是说我们要在讨论“东西”之前，必须先说清楚这些东西究竟“是什么”。“是什么”在逻辑学中，就是概念定义。

清晰的逻辑有助于我们分析事理，但并不见得能让我们发现“事、情”的端倪。逻辑中的“事”必有序，“情”必有势，然则我们现实中的事情大多是处于一个无秩序、无态势的境况下，以及处于这境况下的一个过程片段之中。因此往往地，“析理”只是经营管理者的一个手段或工具罢了，它能解决一些问题，但远非全部。

这也带来一种“状况”，如我常常说的：真正的管理者，是不会看管理的技术书的。但是，正如把管理当“技术”来讲是一个强调逻辑的极端，在另一方向上的极端，则是把管理当成

权谋术来讲。“管理学”总的来说分成相当多的流派或风格，不可否认的是，“技术工程”与“权谋政治”都是其中的典型方法。但——尤为重要的是——这样的管理学派本身，也只是把管理当学问来讲，求学者也只是当成技术来学。

这仍然是逻辑思维的问题。我们总得找个确指的东西——例如流派或风格——才能讲清楚我们“怎么管”。然而事实上的管理中，我们面对的是一个确指的背景、一群确指的人，以及一些确指的事。因此具体到管理者，无论是什么流派风格，亦无论是什么方法工具，常常是拿来就用，听人家说好便是好的。若是按照“逻辑一些”的做法，大概应该先搞个样本，做个试验，出个统计数据，并证明一下有效性，然后再实施。但如果真的这样“很逻辑”地去做了，大概是什么结果也不会有的。

具体做事，毕竟不是做学术讲原理，而只是在“解决问题”。谁都明白，管理学跟数学是不同的。但不同处究竟在哪里呢？在我看来，对于数学来说，特定的过程总会产生特定的结果，函数的可靠性就在这里，计算的可靠性也就在这里。但对于管理学来说，即使“特定的过程会产生特定的结果”，这个过程本身，也只是可选的。

“特定的过程”是一个工程问题，工程是一种可选的、实施的、做事的方法。

如上，我们用“还算有点逻辑”的方法来说明了管理本身的逻辑缺失，或者说管理实践中“不太科学”的某些方面。然而，更进一步地，事实上的管理者并不会关注这些推论与逻辑叙述的过程——大多数情况下，他们是“意识冲动”型的。

行动可以去规划，决策可能会慎细，但“事、情”的开端呢？

第一章

任人治事：组织行为的基本认知

孙膑是一个策士，献计给田忌将军，一场赛马赢得了齐威王的千金赌注。孙膑最后成了兵法家，田忌逃楚，齐国名列战国七雄之首，故事讲到这里便成了终局。

马呢？

一场竞技，成事败事的都在那几匹马，但他们没有在历史上留下任何东西。除了有一个人曾经认真地观察过它们——司马迁的《史记》在写孙膑献计之前，还写道：“孙膑发现他们的马脚力都差不多，可以分成上、中、下三等。”而正是因为这种观察，孙膑胜了，成了诸子之一。

工程中的各个角色，在工程或成或败之后纷纷散去，成为下一局赛事中的、终将继续被遗忘的行动者。那些指挥行动的人，拍着脑袋，说着畅想，用着败了局的失败手法，或学着胜了局的得胜策略，试图挽救一个又一个所谓伟大的工程项目。

但是，马呢？

田忌不是因为一个策略而得胜，孙膑不是因为一个策略而得名。一场赛事或者一个项目成败的关键之一，在于对角色的观察——了解他们，知道他们该做什么，能做什么，以及能怎么做。但像孙膑这样了解他们的人，却寥寥无几。

第一节　刺秦与灭秦

一

张良是位儒生，手无缚鸡之力。他找来一个大力士，做了一个百余斤重的大铁椎去刺杀秦始皇。结果因为力士无法分出秦始皇坐在哪辆车中，把铁椎扔到副车上，失败了。后来力士被缉杀，张良逃到下邳，得了侠士之名。再后来，张良追随了刘邦，成了一代名相和谋臣。

张良活着，这很重要。

先求活而不是志死，是面临工程失败的第一选择。无论工程做得怎样，人还在，团队还在，就总还是有机会的。即使力士死了，再找一个，又死了，还可以再找；即使团队散了，再建，再散了，还可以再建。只要工程还在，就还可以有张良，也还可做张良。做张良的第一要素不是聪明，而是活得长，活得长就有机会；做工程的第一要素，也不是做得多好多漂亮多有成就，而是这个项目还在，这个产品还活着。

所以张良的目标一直都在。只要有这个目标在，今天可以找力士去刺秦以推翻暴政，明天可以随刘邦夺天下以立汉朝。总而言之，只要目标在，做什么事、用谁来做以及怎么做，都是方法问题。工程中也是如此，不要以为只有“程序员写代码”能做出产品，买来的也一样是产品，抄袭的也一样是产品，挖了人家的整个团队来，也一样拥有了产品。这些事例，在我们这个行业中全都是有的，是真实的成功案例。所以具体到一件工程决策，只要不违道德，就没有可耻；只要不违法律，就不算错误。

工程的目标明确，做事的法子就有权变；目标不明确，做得再多，力量再大，团队再好，也一样是把大铁椎扔错了地方。

二

出主意的是张良，做事的是力士；主意出错了，力士死了，张良活着。

力士为什么死？

不是笨死的。你周围没有一个人是笨人，大家都聪明。力士大概是肌肉发达许多，但这并不代表他头脑简单。也就是说，力士可以选择跳楼摔死，但绝不是因为他笨才跳楼。力士这样选择，总是因为他“认为这种选择正确”。正确与错误——亦即是非观，是一个人的基本性格组成。若一个人是非不分，那么，大概他也不会成为公司一员、坐在你旁边或进入你的团队。

你看起来“是非不分”的那个人，可能只是他判断“正确与否”的依据与你不同。

程序员所认为的正确，总是一个问题的最优解——这与程序员接受的职业训练有关。若程序员认为系统“有了”一个目标，就必将认为这是“唯一的一个目标”。若不是这样，他就会再努力寻找、一直寻找，因为他认为若不唯一，则必不是“最终”目标，而追求最终目标而不是阶段目标才是程序员的兴趣之所在。这一点倒是很像当年一挥手，便让三千童男童女入了海的秦始皇。对于秦始皇来说，唯一正确的解就是长生不死。

秦始皇很一根筋啊。这没有不好，这反过来其实也很像程序员：程序员总是在刻意的追求中“无意间”创建了一片新大陆。如果没有这样的刻意追求，程序员可能过不了今晚就决心改做市场人员、管理人员或是产品经理了；如果没有这种无意间的收获，那么我们也就看不到 Ajax 或者 Linux 这样的东西了。

有些东西是从无意间开始的，只是努力的追寻让它变得有了意义。有些东西，则是还没有开始追寻就夭亡了的，例如力士的刺秦。

力士相信刺秦是正确的。这是张良唯一能让力士相信的事情。张良设立了一个目标，并成功地让力士相信了这一目标是唯一并最终正确的，这也是力士志以赴死而非跳楼的原因。而力士，在这个问题上并不聪明：需知“刺秦≠灭秦”，这两件事所指的其实是两个不同的“秦”的抽象啊。所以这至少证明他缺乏一个好程序员的核心素质：抽象概念都分不清，当然做不好程序员。

不过，也许“刺秦”的确是力士的目标也说不定——力士嘛，怎么会想着灭秦呢？没必要的事情嘛。

力士死了，没有名字；秦灭了，不是因为刺秦。

三

灭秦是一项工程。若只以结果论，这项工程的总监是张良，老板是刘邦。

张良刺秦王，是用错了法子；张良找力士，是用错了人；张良用力士刺秦王，是设计错了产品。总之，张良最初是错得一塌糊涂的。他比其他的糊涂人要好——至少运气要好一点：他活了下来。

所以成功还是多少要有点运气的。

不过接下来就不仅是运气了。当“灭秦”成为了产品目标时，天地就宽阔了。比如说，秦始皇死不死其实是不要紧的，只要秦国灭了就好；又比如说，秦国灭不灭其实也不要紧，只要不是秦国统治就好。前者是西汉之所以建了朝代，后者是南宋之所以亡国的原因——所谓一个工程的成败，于当时而言，就是定义的不同；于长期而言，就是一个收效的问题。

当然，张良还得跟对了人。据说张良见刘邦之前，是打算追随楚王景驹的，但正巧路上遇见了刘邦。但仅仅“巧遇”还是不够的——就好像说你在公司里被“碰巧”分到了某个项目团队，这样的偶然事件其实是不能决定你能否成功的。

张良之所以追随了刘邦，成了“刘邦的张良”，是因为刘邦对张良“言听、计从”。首先，对于一个团队——哪怕是仅仅两个人的团队——来说，“听别人说话”是第一等级的待遇。我发现许多项目经理失去了威信，倒不是因为他多笨多傻，或者太聪明掩盖了别人的光芒，反倒是他太过刚愎自用，不听别人的意见。于是团队里的声音就渐渐少了，腹诽渐渐多了，隔阂也就渐渐出来了。所以，“听别人说话”倒是学做刘邦的第一要务。

其次，张良是谋臣，谋臣的“谋”是他的生存之本，所以“计从”是他愿意呆在刘邦身边的根本原因。若刘邦只听他说话，而不采信他的计谋，那么张良也呆不久，因为这样的张良便只是一个言官而不是一个谋臣。但是，“计从”这件事情有很大的风险，谁能做到“算无遗策”呢？

所以做好刘邦，也不是件容易的事。

四

刘邦的手边大概有两类人，一类是张良、韩信这样的谋臣与将帅，第二类则是许多的力士。谋臣与将帅大多都留有名字，当然刘邦也留下了。力士却没有——这一点算是遗憾，不过整体上不要紧，毕竟力士很多，做个大一些的纪念碑，能写多少名字就写下多少，这就好了。

力士们要用得好，最重要的是给他们一个目标，例如刺秦；而且要给他们一个靠得住的理由，比如刺了秦大家就有饭吃。当然，仅仅给个理由是不足够的，比如“有饭吃”这个理由就并不充分，因为别家“刺秦公司”或“刺 X 公司”也可能给饭吃。所以当有很多家公司用这个理由时，“目标”就不紧要了：目标因为理由而存在，如果理由不充分，则目标就不是那么必要，这是一个力士们都可以推论得出来的逻辑。

所以有两种公司层面的选择，一种是给出更多的理由，比如更高的薪酬待遇；另一种呢，则是改变他们的目标。后一种选择涉及到两个关键问题：其一，人是利己的还是利他的；其二，人是先利己的还是先利他的。本质上来说，这是对人性的质问——无论是程序员，还是力士(1)。马斯洛提出的“需求五层次”理论对于这一问题的回应很简单：人是既利己也利他的，但总的来说是先利己后利他的，终究来说还是利己的(2)。

简单地说，要吃饭、要活命是最基本的生理需求与安全需求，这是薪酬待遇可以解决的。接下来是在此二者之上的社会需求，这是一个社会角色——首先是职业角色的需求，所以在公

司层面的选择就是职务的提升，以及给出职业发展空间，等等。其四，职业上的尊重感一定程度上来自于专业性和必要性，亦即是说，公司强调职业价值与专业价值是对大家的一种基本肯定，这与“言听计从”是等效的。最后，大家——整个公司、团队中的所有人——都追求的、自我的、发自利己性的需求，便是剩下的成就感了。

无论力士之路，抑或张良之道，其实都是如此。力士将成就感定位在刺秦的成败，张良将成就感定位在灭秦的成败，这仅是目标的不同，其背后的需要——成就感——其实是一样的。当然，张良的道路走得慎重得多：藏身下邳，是求生存；得刘邦重用，是求尊重；灭秦兴汉，是谓成就。

力士连生存问题也考虑得不甚了了，所以刺秦一失，便身亡了。

五

刘邦毕竟是用“灭秦”统一了力士们的目标，无论如何，他做到了这一点。

但是谋臣与将帅，对于刘邦来说，就不是设定“一个目标”就可以的了。这其实也是许多公司留不住中层的真实原因：目标若只是一事一物，那么该事物被替代之日，便是去留取舍之时。大公司死的产品多，那中层也就换得勤：要么换产品线，要么就换公司。总之，如果将“一个产品”作为目标，那么中层就总在失望中煎熬：熬得住的像油条一样在不同的产品线上翻滚，熬不住的换一家公司继续翻滚。大体如此。

反过来说，刘邦若是大公司的老板，就决不会将“一个产品的生死”作为中层的目标。更多的情况下，这样的产品只是老板权衡属下的砝码。这便是所谓“重结果也重过程”的管理思路。从公司层面上来说，中层所必须确保的是对“企业目标”的认可，其次才是“做好当下”的态度与方法。有了这一前提，不因一时一事之得失而奖惩——这样的管理行为才是可以理解的。

这是“有趣的”公司管理制度：成功了十次的人，可能比不上对公司“一贯忠诚”的人。正确吗？答案是：正确。

因为在中层管理以上，最大的成本其实不是一两个产品的得失，而是信任成本。比如说：刘邦为什么信任张良的一个“计谋”？如果刘邦需要花许多许多的方法去考察，又花许多许多的战役去评估，再花许多许多的时间去推演张良的“计谋”，那么张良对刘邦也就失去了价值。张良之于刘邦，大多数时候的核心价值在于“军师是可以信赖的”。只有有了这一信任基础，高层的决策才能迅速而卓有成效，其结果才能显著，其实施方才可行。

一个扯皮会议的背后，不是所谓的缺乏决策，本质上是缺乏信任。

你能说清楚你在会议上的一个建议的所有“企图”吗？

解释很累、很花成本。

六

所以，做好张良与韩信的诀窍在于：取得信任。

很多中层天天要授权，但却没有反过来想想：我真的能放心让老板授权吗？嗯，大多数中层要是反思一下这个问题，大概就得脸红，就得吃不下饭，就得挂靴退场。反过来说呢，大多数没有挂靴的人，主要还是吃得下饭、脸不会红，以及或者不会反过来想。

建立信任的方式有很多种，例如亲近感。我第一次为“亲近”二字感到恐惧的，大概是来自一次淘宝交易——当我打开“旺旺”与卖家初次交谈时，对方的一句“亲，有什么可以帮到您的？”立时就让我惊愕当场。多么廉价、直白而又脆弱的亲近感啊！你千万不能指望如此亲昵而又随意的称呼会给你带来何种信任。如同今晚请你吃饭的老板或同事拍着你的肩膀说“兄弟，我们就这么干了”——这一类的亲近大抵都是靠不住的。

用利益构成的信任也靠不住，例如薪酬。但有种情况是一个特例，即是说“如果某种利益是特有的”，那么这时信任可能要强得多。诸葛亮之所以跟随了刘备而不是曹操，更可能的原因是刘备能给的“利益”更为特有一些：曹操帐下人才济济，相比之下刘备更需倚重诸葛先生。比较来看，薪酬这种利益就没什么“特殊性”。因此你开出再高的薪水，员工该辞职的还是要辞职，对手能挖角的还是照样挖。靠利益构筑起来的团队关系，要多薄弱就有多薄弱——但是老实说，这也比“亲近感”带来的信任要好一些。

薪酬解决的是人的第一、二类需求，即生理需求与安全需求。在一个成熟公司的人力结构中，这是应当被基本满足的——也就是说，它应当不成为主要问题(3)。但是如果它被有意地“放大”成为主要问题，比如说公司就非得宣称“我们具有业界一流的薪酬”，那么无异于把信任建筑在利益之上，这一利益崩溃之日，便是信任瓦解之时。

许多公司在第一、二类需要能基本被满足时，将信任关系“转移”到其他方面，例如认同感。譬如认同公司的目标前景、认同公司的价值观，以及认同“这群人”——这些的确相对明智得多。但是，这事实上正好是违背人性的——我的意思是人的利己性。人的自利性决定了“被认同才是需求”，而此时所谓“认同”却是付出。当一个信任关系建筑于个人的付出时，它也不是那么牢靠的。例如让员工“认同公司的目标前景”，那么授予期权股票吧，而当上市无望或股票贬值时，这种认同就不存在了；又例如“认同公司的价值观”，那么当个人利益与公司价值观冲突时，这种认同就被质疑了。

所以作为中层，你向老板表示亲近，或表示对薪酬的满意，或对他们认为正确的东西表示认同等，都不足以取得他们的信任。作为你的老板，他比你更清楚：将信任构筑在这些东西之上，是靠不住的。反过来说，如果他们因为这些而向你表示信任，那么你也要小心了：无论老板真是猪头，还是他们别有所图，你都得快点换个东家。

七

战国。魏国。一天，一个战士身上长了脓疮，吴起将军前来探病，亲自用嘴为他吸出了脓血。这个战士的母亲听说后，伤心地哭了起来。邻人问她说：“吴将军关心你的儿子，你应该高兴才是，为什么要哭呢？”战士的母亲说：“过去，吴将军曾为他父亲吸脓血。他父亲感动不已，打仗时拼命冲杀，死在战场上。如今吴将军又为他吸脓血，我真不知道他又会死在哪

里！”

战士死没死，我不知道。

吴起的法子偶尔用用是可以的，但想想伤病士兵的数量，就知道这个法子没有普适性。所以如果士兵也这样来想问题，就知道“等着吴将军来为我吸脓血”大概无异于等天上掉馅饼。这个法子，之所以“偶尔地有用，却又被常常地提及”，是因为这是用人术的典型示例。但这并不具有团队价值，简单地说：无法推及到整个团队的方法，都只是技术技巧而已。

吴起要爱兵，好的法子是爱部下，并让部下爱兵；吴起要用兵，好的法子也是用部下，并让部下用兵。“吴起→部下→兵卒”是一个组织结构，找到让这个组织结构活起来的法子，才是将帅之本。若非如此，脓血是可以一个个地吸过去，但临到战场上，却还是只会让当兵的送命。

兵之于将，在于求活；将之于兵，在于求胜。常胜，大家才有活路，队伍才不会散了。所以会不会做产品，会不会让项目活下去，会不会让工程或产品线成为公司关注的焦点，才是兄弟们跟着你的缘由，才是他们求活的普遍法子。

吴起爱兵，并不是他成为将军的原因。

这一点许多人误读了。

八

信任要么来自口碑，要么就来自于实事；所谓口碑，追究起来，也是实事的效应。所以总的来说，所谓信任，是做人的境界，亦是做事的效应。于一个职业角色而言，做到“信任”并不复杂：其一，取得老板的信任，关键在于“做该做的”；其二，取得下属的信任，关键在于“该做的都做得成”。

我常常说要“专业”，便是这个意思。

我必须先摒弃一种所谓“人浮于事”的组织。在这种组织中，“求做事”并不是第一位的，其第一位的是在于“求利”，第二位的则在于“求活”。所谓求利，就是大多数组织角色都在谋求自己的利益，而非组织整体目标的利益；所谓求活，就是大多数组织角色都在为自己能呆在这个组织里的、组织角色上的时间长短而担忧——其一，呆得更长则获利更多，所以求活也是求利的一部分；其二，组织的，以及组织整体目标的价值与“我”是无关的，因为我的目标便是从组织中牟利，或与组织谋利。

总的来看，在这样的组织中，“做事，或不做事”都是次之又次的事情。“专业”在其中是没有价值的，因为没有事可做；即使有事要做，也只是为了延续组织角色的时间；即使要做好，也只是为了“做好这件事本身”之外的利益。所以，人浮于事，因而不求事业之专。若再换言之，便是“人浮于利”。

所谓良禽择木而栖，便是选一个组织以容身的事情。要么你可以改变组织，要么你可以容身

于组织，要么你可以远离组织，这些都是你的选择(4)。你的“专业性”本身，同组织的形态与既定构成是无关的。但你选择了怎样的组织，决定了你的“专业”是否适用于它，这也进一步的决定了你的、团队的、项目的前途与未来。

第二节　看到别人能做什么

一

齐威王用了三匹马，以及一个田忌将军。田忌用了三匹马，以及一个孙膑。孙膑用了三匹马，以及一个计谋。齐威王想得的是快乐，赛马的胜败是无所谓的，开开心心地与大家同乐就好；田忌将军想要的是齐威王开心，如果只能败不能胜，齐威王的游戏就乏味了；孙膑想得的是胜，因为只有得胜，才会得重用。

齐威王事实上连自己有多少马都不见得清楚，何况它们的脚力；他只是想着该与田忌将军赛一回马了。田忌将军不是因为不聪明而不关注马力，而是因为他更多地关注齐威王的心情。孙膑关注于马力，是因为马的成功就是他的成功；他只能胜，只有胜可以选。

齐威王要用好田忌将军，他做的事情是与将帅同乐。田忌将军要用好孙膑，他做的事情是言听计从。孙膑要用好马，他做的事情是观察马的脚力。

大家都做对事，就成功了。

二

作为马，了解自己的脚力是很重要的。

大多数程序员“一贫如洗”，所能倚仗的就是他们手中的技术，如同马倚仗它们的脚力一样。但这种“贫”只是指种类的多少，而不是数量上规模。举例来说，一个很资深的程序员与一个刚入门的程序员，可能在这个问题上是一样“贫”的——唯技术可恃。

与世界上所有职业角色一样，程序员可能会认为自己是“优势物种”。一方面，这一认识可以细化到具体的分工，例如前端认为后端简单，知识结构单一；后端认为前端只会玩玩脚本，数据结构与系统性能等都全然不懂。另一方面，这一认识也会被推及到整个组织中的所有角色，例如程序员可能贬低产品经理，认为他们根本就不知道产品有多么复杂多么难以实现；也可能贬低项目经理，认为他们除了会看墙上的钟，就只会看手上的表，再或是就只剩下看桌面日历上弹出的提醒；也可能贬低各个部门的总监或职员，理由更加简单——他们什么也不懂。

支撑程序员做这件事的，正是他们唯一剩下而又唯一倚仗的东西——技术：一方面，技术是

他们信心的来源；另一方面，技术也是他们实现一切的手段。所以，当公司提出“我们能不能减少程序员”的时候，程序员的想法是：开发一个用于减少程序员的程序——当然，前提是增加几个程序员。类似地，程序员对于新产品的想法是“我们可以做一个”，对于大系统的想法是“更多的人手”，对于稳定的想法是“重构系统”，对于性能的想法是“优化代码”……这些，都可以归结为一个简单的描述：更强的脚力。

程序员超乎寻常地追求逻辑性。例如，他会在会议上纠正你的一个措辞，即使这个措辞并不影响大家的理解——但是，就程序的逻辑的必要性而言，不正确的概念会产生歧义。程序员愿意在会议中牺牲掉十分钟来解释一个名词概念，因为他并不在乎“达成会议的原始目标的必要性”，他更多的在乎“逻辑的必要性”。

程序员热衷于确定一件事物，而反对随意性以及“看起来不必要的”尝试。无论如何，只要一件事情有着可度量、可预知、可确定的结果，那么程序员就对此信心满满——即使这些仅仅是基于他“专业性”的预估。更深一层地分析，之所以程序员敢于拍着胸脯说“这个功能我们三天就可以做完”，其本质原因在于“他希望这一开发过程是确定的”，并主观地加以了确定。确定性是程序员进行“分析活动”的基本需求，他不得不先满足这一点，进而才能论及其他。有趣的是，如果你告诉程序员“这件事一辈子也完成不了”，那么通常会有一个结果：他什么也不做，直到他证明了上述观点的对错。

逻辑与正确是程序的基本思维方法，这些思维方法影响了（大多数的）程序员的性格形成。例如他们不愿意做多余的、重复性的、无意义的工作，也对冒险性的尝试充满了敌意。最特别的一种情况是，他们会说：这不是错误，而是正确的设计。但是如果从基于人性——而非逻辑——的角度上来讨论一下正确性问题，那么：

不能用的，就一定是错的。

三

逻辑是因为其必要性的存在而存在的。

田忌赛马中的胜负并不一定是由马的脚力决定的。即使田忌的马都比齐威王的好，他也不能场场都胜；就算是胜了，最好的法子也是把马献给齐王。所以，“赛马以取胜”这一逻辑，是因“若与齐王赛马取乐，则不能常输”而存在的。一旦田忌将军是常胜，那上面的逻辑必要性也就不存在了，那时“赛马求败”反倒是可能的逻辑。

架构角色往往追求系统整体的必要性，他因此必须预见到“何谓系统整体”，并判断出系统整体的、核心的目的。以田忌赛马而言，系统整体是田忌与齐王双方，目的是使君臣齐乐，最后才决断出“取胜”的必要性。以具体软件系统的架构而言，架构师必须判断的是哪些地方该考虑性能高低，哪些地方该做功能增补；哪些地方宜先实施而后优化，哪些地方又必须先找出最优解再做实施；哪些地方可以通过产品选型来购置，哪些地方必须自主研发……这些都是架构在系统的组织结构上要考虑的事情。但如果相同的事情交由程序员来判断，则答案可能就缺乏了轻重缓急的权衡；交由项目经理来判断，则答案又可能偏离系统的事实而难于实施。

架构是最没有安全感的角色。在架构师的眼中，一切都可能存在，也都可能湮灭于一霎：千日之功，亏于一篑。之所以架构师对任何一个选择都没有绝对的信心，是因为架构这一角色对系统提供的总是“可能性”，而非必然性。从这一点出发的架构师有三种。第一种是殚精竭虑型的，这种人总会认为系统的一切与自己都有关，也将一切的安危系于自己一身；第二种是置身事外型的，这种人总认为系统的成败其实只是系统自我发展的结果，而架构师只是系统的观察者、考量者与守护者。以上两种架构角色其实都过于极端，对系统的发展都是趋害的。第三种架构师则会在这二者之间觅求平衡，他应知轻重缓急，熟稔何时把握什么，争取什么，以及推动什么；他还应知刚柔曲直，攻守并济，把生死看成系统的第一要务，把时间看成调节生死的阀门。

好的架构师会在系统出现急难之前就对你发出警告，更好的则已经提出过解决方案，再好一些情况下，架构角色已经要求你与团队一起预演过这样的方案了。但是一旦急难发生，你需要的是技术高手，是实施团队，是那些在代码中摸爬滚打的程序员。因为这时候再谈架构，无异于房子着火了却仍在埋怨“我们没有安消防栓”。若“着火了，且没安消防栓”已是既成事实，那就应该加大水的输送或鼓动消防队员奋勇争先，尽一切可能去灭火，这才是上策。我们可以追究一个架构师在“如何部署消防栓”和“没有提供着火时的应急措施”这两件事情上的失职，但这种追究或是在事前，或是在事后，切不可以事中。

架构的危险性也正在于此：架构师的失职是无法补偿的。架构师只有尽可能全面地评估系统中“所有可能的”问题与方案，并尽可能使团队与系统处于安全边界以内。退而求其次，危机一旦发生，架构也要保证提供可能的应对之策。

架构师要做的事情，任何一个角色都可以去做，但架构师所需的素质，却在“做”那些具体事务的能力之外。这并不表明架构师眼高于顶、手低于足。须知架构这一角色必备的是眼界与敏锐：放开眼界，才能看得到危机；切中利弊，才能控制系统代价。我们可以要求架构师同时是行动派，但架构师必须控制自己不要变成冲动派。

行成于思，毁于随。

四

事实上，程序员讨厌架构师的未雨绸缪。而且同时，管理者也不喜欢这样。

有趣的事情是，在大多数情况下，在危机出现之前是没有人相信危险的，并且也是没有人愿意为此投入资源的。因此大多数架构师提出的“方案”都可能面临实施者的质疑：那个架构师的想法是多余的、没用的、空中楼阁。没人愿意为这些“可能的异常”付出代码、时间与防护性的子系统。

我至今所知道的唯一一件能让程序员——以及包括项目经理在内的整个团队——所接受的预防性实施技术，大概就是“结构化异常”（try…catch…finally）。但仔细地探究也会发现，其实越是在项目中面临过工期拖延、系统崩溃以及未知的 BUG 的折磨的开发人员与团队，就越能接受在代码中出现更多的、貌似无用的异常捕获。所以看起来，真正的问题并不是程序员以及管理者不愿意推进“预防性的”架构，而是在于：一是他们无法感受到反复的系统

危机所带来的痛苦折磨，二是他们无法找到有效的、可实施的方法来应对上述危机。

层次系统是极有可能符合这一要求的架构方法。但是就程序员与管理层所能接受的观念来讲，层次系统是过于复杂而随意的——只要会说“增加一层抽象”的，就都成了架构师。但正确的层次抽象是确实可以隔离复杂性的(5)，而且通过接口设计可以进一步减少、屏蔽或确指层间的依赖。但这些工作必然带来程序员的不满，因为接口的抽象与使用是多余的工作；也必然带来管理层的不满，因为“层次”这一系统抽象在事实上也约定了组织结构的规划。

测试驱动开发过程是另一个视角下的解决方法。从工程过程的角度上来讲，它通过明确系统的“问题”、“关键点”与“指标”等来将系统参数化，并将这样的参数作为过程中贯彻始终的标准加以推进。尤其重要的是，它事实上也内在地要求了系统构件间的“接口化”与“复杂性隔离”，因此它可以与层次系统有着相当良性的配合。

迄今为止，我没有看到过一个更好的、更简单的“架构”，能切实地应用于项目之中，将可能发生的危机阻隔在一定的范围之内。我在这里所说的“架构”是指两个方面，其一是架构师对系统的抽象方法，例如层次架构方法；其二是指由“软件架构”与“软件开发方法”配合所产生的整体效果，例如层次系统与驱动开发过程的配合。因而，我所指的“切实有效的架构”可以简单地描述为：

目标系统、架构方法与开发过程协调统一的整体。

事实上，我认为只有这样（包括组织的、过程的、实施的等在内）的架构才是对“预防性”的整体求解(6)。

然而，这带来了很高的组织成本。管理者喜欢简化问题，喜欢尽快投入行动，喜欢尽快看到结果——在这三个方面，管理者能很快与实施者达成一致，但与架构师则很难。追究这一问题，事实上症结在于架构与管理二者的相互脱责。首先，架构师所要做的事情，就是推进这些“看起来多余”的事情的实施。架构师若一力担之，便出现了架构师领导下的应急小组；架构师若置身事外，整个团队失去了在这件事情上的基本动力。所以，架构师必须在“方法”问题上迎合管理者，必须使架构的具体实施方法简化、可行、有效。

因此，真正有效的架构方法，就是让整个团队都把这些“看起来多余”的事情视为系统实施所必须的一部分，即“系统化方法”，亦即是团队可以应用的法子。若架构师的主意总是“我们需要一两个高手来做这件事”，或“我们需要一两个突击队”，那这并不会影响团队整体，对管理者来说也必是“可选的”。大多数情况下，管理者只会在不影响团队整体资源的配给时，才会同意这些想法。

而反过来，如果一个方法是“系统化方法”，它影响到团队整体的投入与产出，并且将深刻地影响到系统的推进，这时管理者才会真正地重视起来。这也才是考验管理者的团队决策能力的地方，即我们——整体地——要使用什么样的方法来工作。因为一旦管理者认为“这不可行”时，这个系统化方法便失去了价值。所谓不可行，本质上来说仍然是“对于当前的系统（团队、项目与产品目标）来说，是难于实施的”。

架构师必须与管理者在“系统化方法”上达成一致，从而在团队、方向、目标以及整体策略

上，对系统施加约束。

五

所谓“管”，就是“官”头上的帽子。换言之，一个管理者若想体现自己的“官性”，那么必然提出种种的管理方法。然而，管理人员的真正价值并不在于“设定做事的方法”，而在于帮助团队找到他们做事的方法。

我仍然记得我在团队中推进“代码格式化”时采用的方法。那时我所在团队有着从不同语言、不同团队转来的程序员，其中不乏有着种种开发怪癖的能人异士（相信我，你也会碰到这样的成员的），于是整个团队所提交的代码风格混乱不堪。尽管大家都提出“统一代码风格”的要求，尽管几乎每个人都认为应该这样做，但是具体决定要采用何种风格时，整个会议室吵翻了天：每个人都有要求采用自己熟悉的风格的充足理由，甚至一对大括号应该占据一行、两行或是四行这样的问题，都成了涉及哲学、美学以及基于某种学派的理论的争端。为了这个问题，整个团队整整两天都无法提交任何有效的代码。然后，终于地，在一个会议上我提出了最终的解决方案，我告诉大家：

（1）我决定，我可以接受任何人的、任何风格的代码。但是，

（2）我决不接受同一份代码中有第二种代码风格。所以，

（3）如果你修改别人的代码文件，那么你必须让该文件中的所有代码使用你或别人的风格；你可以选择任何一种，但结果必须保证该文件中的代码风格唯一。

很快，整个团队的代码风格统一了。对于有美学追求的人，我告诉他们，同一个代码文件中有两种风格，肯定是不美的；对于有哲学追求的人，我告诉他们，你若要修改别人的代码，就有责任使代码变得更好，而不是更乱。于是，所有人都在“改代码风格”上付出着努力，适应别人或是改变自己，一段时间之后，他们便找到了最简洁和有效的法子：

能保证最快修改代码的风格，才是好风格。

六

有两种与“做一个官”不同的角色定义，可以更好地实现管理价值。

领袖（leader）是一个极需个人魅力的角色。与大多数人设想的并不一样，领袖其实是最需要那种事无巨细亲历亲为的人来担任。领袖角色最富于行动性，当组织确定一个目标时，最先听到的声音应该来自领袖：

“我来做！”

这种声音具有强烈的号召力，自信果决，有着一往无前的勇气。同时，领袖角色并不是一个孤高自负的人，他深知团队力量远远强于个人力量。但是他并不乐意与平庸合作，因此得到他首肯的唯一方法，只能是“超乎寻常地优秀”。领袖角色最怕拖泥带水的人、事与环境，

他从心底里拒绝自己陷入泥沼，从而避免自己成为悲剧英雄。领袖角色有“不成功，则成仁”的勇气与心态，但他会把这个“成仁”的时间放大到自己可控的范围内，一旦时间失控，他最可能的选择是黯然离场——唯一能击倒他的方式只有“时间”二字。

争取领袖角色的合作是有条件的。其一，他需要正确的方向或是明确的目标，对于前者，他有探索的勇气；对于后者，他有实做的能力。其二，他需要最聪明的人，他可以驾驭聪明。其三，他需要精练的决策团队。他的个性特点，决定了这三个条件是他在团队管理中所必须的补充。首先，领袖角色是行动派的，他是推动系统向前发展的核心力量。但是如果系统的方向或目标并不明确，那么他的行动就失去了理由，他的判断就失去了依据，他的勇气、信心、能力以及种种努力就都失去了存在与施展的舞台。一旦领袖角色失去了方向，那么整个团队将迷茫停滞，时时处于溃散的边缘。其次，领袖角色对于驾驭聪明有着浓厚的兴趣，这是领袖角色管理艺术的核心：领袖角色是彻头彻尾的实干家，因此他事实上是在寻找“有用的聪明”——对“聪明”真正正确的判断应落脚在对于系统的影响上。最后，办公室政治、人际纷争之类是领袖角色会尽力避免的，在这些边角上的聪明只会令他反感。凡是与系统目标无关的决策者、决策过程，都是领袖角色大刀阔斧砍削的对象。

上述条件也反映了另一个事实：领袖角色将组织“适合自己的团队”。因为上述条件中有目标、有人、有做事的方法，这些构成了一个独特团队的基因，也是这个团队持续的、长久的价值核心所在。这也表明，领袖角色在“组织授权”方面是有条件的——他需要更高层次组织（例如公司、事业部等）的明确授权。

第二种角色称为 Owner（主管）(7)。对于一件事情，一个 Owner 的正确反馈是(8)：

“我负责！”

Owner 最需要的是清醒的头脑、务实的作风，以及敢于牺牲的精神。首先，他需要足够清醒以确保在合适的问题上找到合适的人、合适的方法，以及判断什么是“合适的解”。例如在技术专长方面，他可以借助团队的力量；在方向决策方面，他可以借助架构的力量；在团队激励方面，他可以借助领袖的力量，等等。Owner 的协调能力是他对整个团队的最大价值，因此他将尽力避免与任何一个角色的冲突——事实上他是尽力与所有人合作，并通过这一协调能力来消化团队内部的矛盾。其次，Owner 角色对“可能的意见”的判断趋于务实，因此可能是“不那么聪明的”，也可能是“不那么完美的”，但对于此时此事、于当时的选择必是最妥贴的。最后，Owner 视系统为自己的一部分，因此尽管他可以接受架构角色关于方向与策略的观点，但他总是更优先地考虑“系统当前的生死”——即使有人承诺给出“更长的时间、更大的空间”，也不能使 Owner 忽略系统当下的现实问题。他总是清醒地面对这样的“切肤之痛”：若系统出了问题，是 Owner 的问题；若系统死了，Owner 便失去了他存在的唯一价值。

Owner 眼中只有系统，“系统是我的”，激发着 Owner 的自利性；“我只有系统”，使得 Owner 舍无可舍。无论从概念上、心态上，还是从组织角色上 Owner 都与系统绑在一起，他担负着系统责任，其存在就证明系统是有价值的，而这也同时就是他的管理价值。

所谓系统，并不一定是软件，也可以是组织。

七

我注意到一个事实：即使是最无能的管理者，在争取部门利益时也是最积极的。

一个人——我是指职业角色——的底限，在于其求活的根本。程序员求活，在于技术，所以苦练技术是每一个程序员都愿意投入的事情。新项目、新产品、新技术，凡此种种，都是让程序员听一下就“连后脑勺都乐”的事情。架构师求活，在于眼光，所以放开眼目是每一个架构师都愿意立即着手的事情。分析领域、跟进系统、参与决策，凡此种种，架构师尽管貌似烦不胜烦，但实则不遗余力。Leader 与 Owner 的求活在于他们所依赖的环境，所以砍掉系统，Leader 就扯旗走人；砍掉部门，Owner 就跳脚骂娘。反之，Leader 总是试图使项目更受重视或优先级更高，Owner 则总在强调资源与规模。

每种管理培训都在谈“成功学”、讲“价值观”，仿佛人人都在追求理想、道德，有着崇高的目标。但现实中往往并非如此，我们的组织并没有这么复杂，组织中的人也并没有这么复杂。大家都只是在守着自己的底限，并“尽量地做些事情”而已。

“尽量地做些事情”，如果正好是组织给他们——这些具体的角色——的目标，那么大家就可以庆幸了，因为这意味着系统必将向良性发展。否则，大家就必然面临个体与团队的利益冲突。而在面临这样的问题时，员工能守着“职业道德的底线”就已经不错了，怎能奢求系统的发展呢？因此只有当他们视这一目标为自己的目标，其利益是自己的利益时，才会一往无前。在这个问题上，人，首先是自利的——每个人都在追求自己在团队中的位置、价值与认可。

但是，正确的做法是先看到别人的位置与价值。因为认可是别人给的——欲先取之，必先予之，若你都不认可别人，那别人自然也就变尽了法子来排斥你。所以，若以团队为背景，则重要的不是先看到自己，而是先看到别人；重要的不是先看到别人什么都不能，而是要看到别人能做什么。

用与不用，在你。

第三节　要做事，不要管理

一

“我凭什么要帮他？”

A 公司很成功地处理了这个问题。一个新员工在 A 公司入职之后，总是会被分派一个师傅。师傅与这名新员工之间，被公司从组织层面上施加了一个名义上的师徒关系。并且，在此之后，这一师徒关系还是一荣俱荣的：在季度考核中，有最佳学生、最佳师徒，以及最佳人师

之类的奖项。

建立并维系一种关系，是一个很好的“帮他”的理由。类似地，还可以有“家族”、“帮派”、“兴趣小组”以及“（同学）会、（同领导）门、（同乡）宗”等繁复的社会关系系统。事实上，我们的组织管理思想，一面在骂着被称为“国粹”的人情关系，一边又在利用着人情关系来维持组织的有效性。作为组织者，我们自己都没有从心底里放下那些自己鄙弃的东西，却又信誓旦旦地要将这些东西踩在脚底。

我们真的找到了那个问题的答案吗？

当我们问对了问题，事实上也就找到了解：

“既然在一个团队中，你为什么不能帮他？”

二

总说“物以类聚、人以群分”，人们若只扎堆在一群里，是没有“分类”这样的属性的。所以若真把项目团队说成是“一群人”或“几群人组成的一大群人”，其实是无法考察他们的共性与差别的。真要达到这一目标，还是得把团队中的人分作几类。

不同分类方法的着眼点是不同的。一种是阶层化的，例如“下者用器，中者用人，上者用势”，是根据不同人在组织中的行为方式来区分的。事实上这也决定了他们的技术、能力与素质的不同。所谓下者用器，一定是“（工）欲善其事，必先利其器”的。——所以我常说“如果你认为这句话对，说明你是‘工’”，因为这原本就是你的工作方法、状态以及你构建知识、能力的基本模式。所谓中者用人，一定是举贤任能、政治教化的，辨的是人情关系，用的是人力才智，察的是人事浮沉，能力多强调在情商方面。所谓上者用势，那么就必然要能深悉事理事态，对形势方向尤其敏锐，在战略宏观上有着超常的判断力与决策力才行。这样的团队构建自然地形成了等级，形成了不同层级之间不同的行事风格与判断标准，因此带来的冲突是不可避免的。在这其中，我们通常谈及的凝聚力向心力之类，多是由人情世故或规章制度来形成与强化的——前者，是中层用人的法子，后者则是上层用势的法子。

另一种分类法基于对人的社会性的考察，根据人们在群体中的自我分工来进行，例如“领袖与追随者”。这一般取决于对自我的认知，以及对他人的认同。自我认知与社会认同是人的基本社会属性，如果一个人缺乏自我认知（缺乏自信或过于自信），或者缺乏社会认同（不信任他人或无法获得他人的信任），那么他都无法在群体中找到合适的位置。类似于“领袖与追随者”这样的定义建立了基本的“人与我”之间的社会角色关系，这种关系既构建于认知与认同之上，也随着它的变化而变化。例如，也许有一天追随者会提出“那些人、那些方向是不是值得我追随”这样的问题，或者领袖基于他对自身在团队中的价值的认知产生了“没有我就没有这一切”的态度。可见，人的社会性（例如自我分工等）并非一成不变，它给整个团队组织带来了不稳定因素。

再举一个分类法的例子，比如朋友与敌人的区别。我们的团队很容易由“朋友”构成，尤其是小到三五个人的时候。这样的团队看起来应该能持久，但真正做下去，却发现矛盾重重，

渐渐地朋友做不成了，队伍也就散了。究其原因，大家都只是冲着朋友的情面儿凑在了一起，却并没有人深刻思考过“如何构建与维护团队”的问题。事实上，所谓“朋友”有两层含义，同门为“朋”，同志为“友”。如果大家原本就是出自一门，比如同宗、同学、同事等，那么就可以算作是“朋”。而这只是基本的、能快速抹去人际隔阂的一种亲近关系，这种关系能让大家快速聚起来并以最少的成本达成共识。创业之易便易在这里——我们总可以找到一群“看起来相同”的人。

但是接下来呢？如果大家在团队中的目的不一致，各自打着自己的小算盘，那么就不是有着共同志向的“友”。所谓朋党、友党的区别也就在这里。朋党有变成友党的基础，但不一定有“成为友党”的事实，因为后者取决于大家的“利益点、目的、方针与蓝图构画”是否相同。而一旦——理想情况下——整个团队中有了一个领袖人物，能在“利益、目的、方针、蓝图”等方面统一大家的思想，那么这个团队才能成其为团队。更有甚者，领袖人物也能敏锐地发现对手的利益取向，通过共同目的或方针蓝图之类的工具来化敌为友。所谓“没有永远的敌人，只有永远的利益”，说的便是这个意思。

团队之中，朋友与敌人是没有那么明显的界限的。这种矛盾的真正出处在于：大家在图什么？统一了图谋（亦即是“共同的志向”），才谈得上谁跟你是友，你跟谁做友。至于“朋”者，倒是简单了，刚从小酒馆出来那几位，不还一边走，一边唱着臧天朔的《朋友》吗？

由所谓“朋友”构成的团队，最终的溃散通常是出自利益的侵蚀。

三

团队是什么？这是一个似有解而又无解的问题，比如我们可以答“一个人以上的群体”，但这样的答案并不能反映团队的根本属性。

换个问题，例如“团队从哪里来”，就会很好回答了。因为通常“你”面对的团队只有两个来源，其一是自建，其二是接手。这里，我们先谈第二种，即“接手一个既有的团队”。注意，在这种情况下，你首先必然、必须——将自己的角色定义为：Manager，而这也确实是最棘手和最容易出错的。而且，老实地说，作为 Manager，你的确是一个“官”，因而你也必须有着自己的管理风格与手法，并行使你的管理责权。

你必须明确一个事实：Manager 的权力可以看做是天生的——因为在接手型的团队中，你是被“任命”的。无论你是“有能者居之”或是“有德者居之”，你以前的能力才干与德行声望，对于现今的团队来说都没什么意义。——你接手之前的东西都化作了权力，接手之后的东西都被称为作为。作为管理者，你（对上或于下）能取得的所有信任，将集中于一点：

你的作为，是否对得起你的权力。

四

你必须先观察/了解这个团队的组成，尤其是其中是否存在两种关键角色，即 Leader 与 Owner 型的人物。我们说过，这两类角色不一定是“官”，但他们都能很好地（甚至是更好地）实

现管理价值。而 Manager 对于他们的了解与运用，很大程度上也决定了自己的成败。

Leader 决定了一个团队的风格与气质，他引领团队的方向并将一大批骨干吸引在自己的周围。对于一个既有的团队来说，Leader 对 Manager 的阻碍主要会来自于两个方面。第一，Leader 确实存在管理欲望。但本质上来说，Leader 仍然是实干型的人物，因此大多数情况下，他的管理欲望起源于对整体环境的不满。也就是说，他既知道“碰触到官僚的、管理性的事务”将是一个泥潭，但他又担心如果不这样做，自己的目标与理想是不能实现的。对于这种情况，Manager 应斟酌考量，如果 Leader 偏重于事务推进，则应该努力使他重掌团队事务；如果偏向于管理协调，则应该考虑是否给他管理角色和责权。但是无论如何，Manager 应该清楚，Leader 的去留对于团队整体来说是生死攸关的。

第二，Leader 努力的方向或试图达到的目标，与团队接下来要做的事情并不能一致。这是一个相当复杂的问题，因为团队更换管理层，多是因为不能达到而放弃旧的目标，或旧的管理层无能。但无论如何，总之是旧的事务已经无法正常推进了。这种情况下，Leader 在“方向问题”上其实已经受到一次重创，现在支撑他的主要理由——甚至是唯一理由可能就是“做，或是不做这件事情”。一旦团队原定的目标改变，而又不能使 Leader 接受新的目标，那么这个团队离实质性的溃散就不远了。对于这种情况，Manager 应该反思自己接受任命后的职责，究竟是从事新的项目还是持续旧的工作。一旦确实要“选择新的方向”，那么除了争取 Leader 的认同之外，可能就只好重建团队了。

Owner 最有可能对你的到来愤愤不平，但也最有可能衷心期待。但这两种完全相反的心理却都出自同一个缘由：Owner 认为自己是系统的所有者。作为 Manager，你必须注意的是：对“系统”的评价，在 Owner 看来其实就是你对他的评价。例如，假设上任伊始，你就说“看看你们这个烂系统烂摊子，瞧瞧你们做的这堆无法收拾的烂事情吧”，那么基本上就打破了 Owner 对你的一切幻想。Manager 对于系统的态度，很大程度上决定了 Owner 对于“合作或排斥”的选择，如果 Manager 对系统表现出热心、关切并且努力推进，则 Owner 会认为“这起码是个来帮忙的”。

Owner 的另一种排斥心理，则来自于个人职业发展中的瓶颈，例如他会想：“为什么不让我来当这个系统的 Manager？”或者会干脆认为：“就是这个人挡住了我成为 Manager 的路！”但这种组织角色上的问题，其实是任何一个 Manager 的职业生涯中必须也是必然要面临的事情。一般来说，你必须表现得比别人更优秀、更强势、更有能力和资源（但是我一定程度上反对“更有手腕”这种技巧性的“处理”）。而且你必须明确一点事实，即：你的权力来自于“任命”，而非“实有”。因此在“自上而下”的管理组织中，你已得到认可；但在“自下而上”的行事过程中，你还远远未被认同。所以，如何应付“Owner 的排斥”，可能是你在团队中表现出管理风格的第一个契机。而且，这也决定了你是否能真正掌握这个团队。一般来说，你可能会有如下几种选择。

（1）如果 Owner 确实有团队管理能力，那么可以将团队的权力还给 Owner，例如保留原有的结构、做事方法以及汇报与审核体系。

（2）如果 Owner 只是团队中的黏合剂而缺乏必要的管理能力，那么你应该表现出强力而又强势的一面来构建团队，并在这个过程中带领 Owner——如果他接受你的领导。在这一选择中，如何保证 Owner 对团队的归属感和积极性是一个很重要的命题。一般来说，信任他，

更好的沟通、商谈，以及将更多的事务与责任交付给他，可能会是一个不错的开端。

（3）如果 Owner 在团队中只是一个老好人的角色而缺乏必要的素质与能力，那么必须先把他放在一边，重新任命一个角色来替代他。注意观察他的情绪变化，帮助他重新认识并找到自己在团队中的合适位置。如果（但这常常发生）他腹诽不已并给团队带来负面的、难于控制的影响，那么你可能必须放弃他。

对于接手型的团队，用人治事，是作为 Manager 的第一要务。

五

我们还得谈谈既有团队成员的普遍心理。事实上大多数不成功的团队是既没有 Owner 也没有 Leader 的，因为如果有（哪怕其中一个），那么这个团队也很难走到现在这种困境，以至于需要你来做他们的 Manager。

第一种心理因素来自于对你的了解与认同。你是 Manager，是官，那么大家首先看到的是你的职务高低——这并不一定是官性与奴性之类的问题，而通常是你的职务高低决定你能把握的资源，进而决定这个团队的未来。你在多大范围内有发言权，有影响力，以及是否有过成功的管理实践，这些都可能是团队成员在你到来前后会多方打听、收集并相互交流的“情报”。

接下来是对你的直观而又主观的认识，例如你所组织的第一个会议，或第一次团队活动，或者第一次约谈。包括时间计划，内容上的组织，你对现有团队、系统所了解的程度，你说话的语气、精神状态以及条理逻辑等，这些都会成为团队成员对你的认识的关键组成部分——一些人甚至会把你的约谈次序都视为具有某种象征意义。必须强调的是，这些主观印象一旦形成，就很难在短时间内改变。因此，除非你真的想形成某种印象，否则你应该特别留意这些在“新官上任”时期的安排。

事实上你对团队的了解并不完全取决于你的观察，在很大程度上是来自团队“展现给你的那一部分”——这基于团队对你的认识，进而取决于“团队会告诉你什么”。所以作为 Manager，你必须知道并预设了你之于他们的基本印象，这既是管理方法的一部分，也是管理风格、素质修养的一部分。再接下来，才谈得上你对他们的了解。这包括对人和对事两个方面。除非你真的了解团队在做的事情，又或者你就是这方面的专家，再或者你此前负责的就是相同或类似事务的团队，否则你千万不要在团队面前“不懂装懂”——就我所知，这是在技术性项目团队中最让人绝望的管理行为。

如果你真的不了解这些“具体事务”，那么你可以找到关键角色（包括技术专家、Leader、Owner 或者某几件事情的“头儿”），征求他们的意见、建议与问题。你必须能足够耐心地充分利用自己的逻辑能力来梳理这些事情。即使你不懂这些事情的细节，你也必须将事情的骨架（这很类似于架构）整理出来，并尽量多地参与团队的沟通而又尽量少地干涉事务的细节。这一切，便如同《大道至简》中说的“跟随蚂蚁，但别掉进蚂蚁洞”——你的目的只是要让自己变得“懂一些”，而不是变成专家。你必须了解“团队在做的事情”，才能对这些事情有所评价，进而有发言权。又如果，你被派来的目标是“放弃现在的事情”，那么上述的过程既是你寻找“给大家一个放弃的理由”的过程，也是你“了解这群人的特点”的过程。所以，无论如何，你都需要先经历这个过程。

这之后，才是管理。

六

团队中不一定都是做事的人。

从统计学上来看(9)，团队中积极做事的成员大约只有 8%，而 40%～66%的成员都是随大流的。基本上来说，积极或可能积极的人占到 2/3，而近 1/3（20%～30%）的人其实是“理性利己主义者”。

对于一般人来说，“做事”是必要的。这缘于两个方面的因素，其一是“大家都在做事”，其二是“我确实想做成些事情”。但对于理性利己主义者来说，“做事”是迫不得已的行为：一方面，不做事就会被一个做事的团队赶走；另一方面，做事可能是必需的谋生手段。但正好因为他们是“理性自利”的，所以这种“迫不得已”最终会变成两个对自我的、理性的拷问——进而争取利己价值的最大化：

（1）如果只是为了“不被赶走”，那有没有比“做事”能投入更少成本的方法？

（2）是不是只有“做这件事”才能谋生？

所以滥竽充数的南郭先生其实是典型的理性利己主义者——既然齐宣王喜欢听合奏，那么在他所在的团队中他就可以什么都不做而不被赶走。同样，如果“赶走别人”比拼命做事（以便让自己留下）代价更少，那么南郭先生也会想办法挤兑走那个发现自己没吹竽的人。但细究起来，这样的南郭先生并不是没有做事，而是通过更简单的法子来达成上述留在团队中的基本目的。而且，一旦“留在这个团队中”变得毫无必要，比如说有了更好的位置，或有了更好的谋生手段，那么他们就会毫不留恋地离开。

对于理性利己主义者来说，不但“是否需要做事”以及“是否可以投入更少的成本”是问题，而且即便“努力做事”确实是唯一之选，他们仍然会踯躅不已。因为还有两个与利己、与价值相关的问题：

（1）为什么要“给你”做？

（2）为什么要“在这里”做？

所以对于这一类成员来说，价值取向最终必然是偏向于私己的。但是从统计学上来说，三个人中就有一个是利己的。因此：

你面临的团队，并不会比这个状况好多少。

七

那么，我们能通过团队教育来改变这一状况吗？

不能。

我并非是在否定团队教育的作用。事实上，我认为团队教育很有意义，但这只是体现在改变技术、协作以及形成一些基本认识方面。例如，我认为团队技术培训很有用，因为如同军队一样，相同或相配合的作战技能总是能得到最佳的效果。又例如，我认为企业文化培训很有价值，但价值并不在于“让大家变成同一种人”，而是让他们形成对这个企业的“共同认识”——基于这种认识，新员工就能有机会来调整自己以适应这个公司的文化与行事风格。

不要忘了，团队中有超过半数的人是从众者。团队教育对这一类成员的影响是良性的，既易于形成好的团队风格，又能使团队文化得到有效的延续——所以团队教育是“自建团队”中相当重要的方法与过程。但这却很难作用于前面所讨论的约占团队 1/3 的理性利己主义者。他们相对理性，因此具有独立的思考与行事风格；他们选择有条件地合作，并期望合作从整体上来说总是利己的；他们算计着与他人交往的收益，并期望投入更少而又获得更多。所以，公允地来说，有这种思维的成员更为理性、更有自主思考的能力。

这些成员并非“不可救药”——准确地说，是他们不接受你所给予的“药”，例如“被教育”。解决理性自利问题的法子是降低收益。确切地说，就是在同一层面上、更为公平的收益分配。在一个做事的团队中，如果收益总是以做事的多少、收效来衡量的，那么自然没空子可钻。

所以我常说：一个好的团队是不需要 Manager 的。因为 Manager 的收益并非来源于做事，而是来源于管理。反之，如果将这一收益方式置入团队中，例如，让“管理”或“在管理上的投资”成为衡量团队收益的标准时，那么围绕团队目标的、项目目标的“做事”就显得不那么要紧了。而一旦实施这种做法，那 1/3 的成员可能就会选择跟随管理层，并通过感情投资或夸大功绩等投机方法来获得收益。由于 Manager 离具体做事的层面通常很远，因此判断一个人的方法往往来自于组织性的考量，例如是否听话，是否好沟通，是否能信任，以及是否看起来“不错”。

可见，搞坏团队的，正好是管理行为本身。

八

管理行为应该从组织层面获得收益（而非从团队中）。例如，我在“愚公移山”中说到过的两个故事，当端木氏献上黄金车乘给晋王时，晋王赐他爵位；当端木氏展现军事、政治才能时，晋王拜为高官。爵与官都是两种组织层面的收益，效用却各是不同。赐爵，是与端木氏的利益互换，有爵位并不一定参政；而拜官则是授了权，是要主理国事的。

从组织层面的行为来说，作为 Manager，你确实有机会将自己变成 Owner 或 Leader——这两种角色才真正地融入了团队。不过，在这种情况下，Manager 还需要考虑组织上的必要性。

温柏格的法子——“让他们签约”是一个简单的能让你变成 Owner 的办法；而他又说“否则离开”，则是典型的 Manager 的失败。“签约”在西方文化背景下很可能是一个好办法，并且从团队构建的方法来说——“约谈”并在沟通中达成共识，也是所谓签约的一个具体的

可操作手法。所以总的来说，这是可行的。

但是如同前面所讲的，若团队有 Owner 或 Leader，那么“你从 Manager 转变成 Owner 或 Leader”的必要性就很成问题，并且这势必会给团队内部带来权力冲突与组织结构上的震荡。反过来，即使团队中还没有 Owner 或 Leader，那么“提拔与任命”仍然是你作为 Manager 的第一选择，而“亲历亲为”则是在你对整个团队及其目标、方向和流程方法有足够的驾驭能力之后的第二位的选择。

如果你不得不选择变成 Owner 或 Leader，那么我必须告诫你：你将不再是“官”。事实上，官性与奴性是我能找到的、对团队伤害最大的两种人性。因此，至少对于这个团队来说，你得收拾好官派的做法与逻辑，准备好以 Owner 与 Leader 心态与能力来应付接下来的一切。

这个团队是做事的。你既然决定给自己一个角色，那么也就必然是一个做事的角色。

第四节　伯夷与叔齐是怎么死的

一

历史上，有两个人很有骨气地被饿死了，这就是伯夷与叔齐。这两兄弟先是相让而不做孤竹的国君，后是相携而不食周朝的粮食。再后来，他们饿死了。

饿死了就饿死了，有人说“这死得很好”，那是跟“死得不怎么好”的情况去比较的。大体上，如果能好好地活着，是没有人会选择去死的。那么，他们又是怎么会死的呢？

首先，伯夷与叔齐认为周武王还没来得及埋葬父亲便要打仗，是谓不孝；作为诸侯要去讨伐君主，是谓不仁；于是这样出兵是不讲道义的，而这样得来的天下也是令人不齿的。所以伯夷与叔齐并非不想活着，而是周朝没有给他们一个——他们认为合适存活的环境。其次，伯夷与叔齐跑去首阳山上采野菜吃，但是有人给他们说：周朝统一了天下，所以粟米固然是周朝的，那野菜也是周朝的啊。于是二人无法可想，便都齐刷刷地饿死了。所以伯夷与叔齐并非一心向死，而是他们自己无法给出自己一个可以求活的逻辑。

既得不到一个可以存活的环境，也寻不到一个可以求活的逻辑，因此伯夷与叔齐便只能这样死去。这种舍身取义地死，当然必将传为美谈——我们需要这样的美，也需要基于这样的美来谈。我们的道德观念、价值取向需要以一些东西作为事例，以丰富我们在这些问题上的种种规约。

然而，没有一个项目会因为死掉而传为美谈。

二

所以公司要给项目一个生存的环境，项目要给自己一个生存的逻辑。

如何构建项目的生存环境呢？

项目的生存环境与公司的项目管理体系有关，例如公司决定如何进行项目立项、审核、推动、考评等流程。这样的生存环境是一系列管理行为的总和，它与具体项目内部的运作方法没有直接的关系，但却实实在在地影响了这个项目的生存与状态。事实上，它也反映了公司对于产品、项目、团队三者的整体态度。

首先，公司是否基于产品建立了一系列的外围组织（或更为细致地说，“产品”作为一个市场化的基础条件是如何被定义、被评价以及是以何种形式作为公司运作的基础构件的），这是后续问题的基本前设。表 1-1 列出了三种需求类型(10)。

表 1-1　对三种软件产品定义及其需求类型的简单考察

下面，我们来分析这三种需求类型。

（1）若需求方是生产型企业（以下称为 A 型），需要的是业务生产系统，那么产品通常是一个软件产品包，开发方需要现场安装、部署、调试以及技术培训；从长期来看，开发方还需要提供版本计划、维护手册、作业限制以及应急方案等。这事实上也包括开发方与需求方是同一公司或公司的不同层次的情况。

（2）若需求方是一般的互联网用户（以下称为 B 型），需要的是一个在线功能，那么产品可能是一个线上网站，所有的产品构件都在开发方所能支配的服务器、机房等环境中运作，而使用功能的客户端可能是网络用户的 Web 浏览器，或者其他第三方网站通过 API 进行的访问。

（3）若需求方是离线的桌面软件用户（以下称为 C 型），需要的是一个能解决具体问题的功能性软件——这包括桌面操作系统、移动设备、手持终端等环境下的软件产品，那么产品的形式可能就是一个离线安装包，以及一些附属的构件（例如在线手册、在线升级或配套的工具软件）。

上述三种类型的产品定义、实现过程和交付方式都是根本不同的。例如对于 A 型，公司对产品的要求就是客户需用、能用以及符合客户的生产运行环境。如果客户现有的应用环境是 Java+Oracle DB，那么公司就必然是基于 Java+Oracle DB 来开发，项目所需的人力及与技术储备就都受到这一前提的限定，项目的评价自然就是产品订单，项目的立项依据就必然是业务人员接触到的市场信息与客户反馈——这些基本设定都依赖客户需求，而与开发方的技术能力与产品设计几乎完全无关。从公司整体上来看，应当是业务驱动型的组织结构最为合理，技术研发上的创新会少，技术实现上也会保守很多。

假定与 B 型来做一个比较，那么 B 型就并不一定适宜业务驱动。因为 B 型的业务部门可能

会碰触不到确切的用户需求，以用户需求为核心依据的业务模式就会发生变化。反之，应从更大范围内来考虑以互联网用户行为为背景的产品推演，并关注基于业务链上下游需求的、功能性（或制约性的）产品设计。于是，在这样的背景下，面对需求问题，业务部门的形态将变成“观察－规划”型，而非（与 A 型类似的、传统含义上的）“收集－反馈”型。所以通常在 B 型产品的公司中，业务部门是以构建产品线为其核心工作方法的。进一步地，整个模式就会变成产品与产品线驱动，例如“产品”的定义就可能是子网站、多级域名或功能模块，项目的评价依据就会是上线下线、运行流量以及用户数量等因素。

所以，项目的生存环境首先依赖于产品定义与产品管理方法。进而，只有在公司内形成与产品过程模型相匹配的项目过程模型，项目的生存问题才会有解。但不幸的是，大多数公司在对待这两个模型的做法上是有冲突的，尤其是在企业转型阶段——请尝试着考虑一下微软与谷歌在产品形态上的不同，以及微软公司在向后者转型中一些可能的障碍。所以，项目组并不一定总在“舒适”的环境中生存。也因此，当公司转型时既可能“砍掉”一些项目，也可能有一些项目自然地就消亡掉了。

产品与业务的规划，是公司为项目准备的生存环境。当我们人性化地来看待项目的生存问题时，这一环境正好决定了项目的价值观、归属感以及项目的基本道德伦理。例如，一个项目是否更好、更值得立项、更有效益，可以看成是项目价值观的问题；而一个项目是否觉得自己重要、受重视以及有前途，则就是一个归属感的问题。至于两个项目间的资源之争，是“掠夺”还是“奉献”，抑或是“有条件地相互合作”，那么就是一个基本道德伦理问题了。

三

对于“铁打的营盘，流水的兵”这句话，很多人认为说的是一个团队问题。

但事实上不是。因为团队总是以人为基础的，人——比如说兵——没了，团队也就没了，所以一旦“流水的兵”是事实，那么所谓“铁打的营盘”就一定不能归为团队问题，也不可能在团队管理中找到解。然而这一点往往会被忽视，于是我们会看到，为了寻求持续、长久的发展，管理者总是将项目与团队关联起来，试图通过“更好的团队建设”来得到更好的、更稳定的项目。然而最终，团队该散的还是会散，项目该死的还是会死。

到底是什么决定了“铁打的营盘”？

军营的体制不是为了单个士兵或将帅而设立的，部队可以拿“雷锋班”“杨根思连”这样的荣誉称号来激励大家，但并不意味着这个班或连就只围绕这两个英雄人物来建设。军营的核心在于战争，训练、操演、技能、口号、人事、后勤、奖惩等都是围绕战争这件事来进行的。因此只要部队的目标还是保家卫国，那么不管士兵换了多少轮，也不管部队迁了多少地方，营盘都是那个营盘，军号定时地响起，连节奏都不曾改变。

项目的确离不开人，但项目的核心并不是人。项目的核心是事。事在、目标在、环境在、实现目标的可能性还有，那么项目就还有机会，项目就还没死——兵，可以再招；团队，可以再建；项目经理，可以换个人来做嘛。所以，根本上决定项目生死的，是决策层说“No”：产品不做了，产品线没了，业务方向放弃了，市场计划失败了……诸如此类。

我所见过的项目，通常会把自己的生存逻辑设计为“资源丰富+需求明确”。这样的逻辑当然是好的，是要去尽量满足的。但是，如同伯夷与叔齐一样，这其实是给了项目一个寻死的逻辑：要人、要需求、要计划、要配合、要……所有需要的一切，要不来就饿死，饿死了大家都玩儿完。所以在公司看来，所有的项目都在喊饿，都有的项目都岌岌可危。

活着的那个项目可能只有一个人，也可能一个人也没有。

今天做不了，明天做嘛。

四

一个项目要做——当然我们不讨论根本不需要做的项目——那么也通常只有两个“相当废话”的理由：一是因为当前必须要做，二是因为将来势必要做。在这许多个项目之中，甲领到了必做的项目，乙领到了要做的项目。于是甲、乙的生存问题便马上对立起来了。甲说“我要人”，乙说“我也要人”，然后齐齐地跑来问公司：怎么办？公司说：你们都很重要，所以一边分一点儿人吧。于是，甲乙两边人都不够，甲很快饿死了，乙慢慢饿死了。

于是，有人说：我们把要做的事分成重要而紧急的、重要而不紧急的、紧急而不重要的、不紧急也不重要的……但只要这样一分，所有的人都立即跑来说：“我们项目是又重要又紧急的，而是还是所有项目中最重要而又最紧急的。”所以，更多人就说：管理要的是平衡；要平衡人，平衡事，平衡……

平衡是求死之道。

五

与其如此，不如告诉大家真相：这个世界没有所谓的平衡。

“要平衡”或“要公平”只是你跑去找老板要资源的一个说辞罢了，无论你做了什么努力，或者老板因此做了什么样的决定，平衡事实上都是不存在的。你得到了资源，本质上就是掠夺了另一个项目的资源。你能谈平衡，只是因为你有资本站到老板面前去谈而已。

项目不是平衡出来的。正确的情况下应该是：该做的就做，不该做的就不做。于是你又会问：凭什么来分“该做或不该做”呢？答案是：不凭什么，那是决策(11)。决策？是的，决策！哪怕只是你看起来拍拍脑袋就出来的决策，总好过没有决策——没有决策力的组织，乌合之众而已。

当然，如果你非得认为做决策的那帮人都是猪头，那么：

你还呆在这里做什么？

六

总的来说，我们需要明确的产品定义，并据此来明确我们到底要做什么事，以及基于怎样的组织体系来做。并且，我们假定那些决策者都不是猪头，因而项目明确，决策可行。那么我们还缺什么？

我们——在整体上——还缺一种做事的心态。《南都娱乐周刊》采访《金陵十三钗》时，张艺谋说(12)：

当年抓这个剧本，为此付出了很多心血，四五年的时间去做一个好作品，希望能成功，希望是个好电影，我从心里来说，真没有具体目标。卖到 10 亿，要得奥斯卡，我没这么想过，我要这么想，就不是导演所应该有的心态，我就是平常心去做，尽管耳朵边有很多这样的话。我知道这些豪言壮语，但是我知道自己的工作是什么，我专心致志于把这个电影拍好了，就行了。

一个团队有一些不同的声音，有一些豪言壮语都是可以的。但做事的人要时时知道自己在做什么，该做到什么样子，并努力地去完成既定的目标。这种心态，就是整个团队应有的、对内的一种环境。当外在的、人为的、商业的、市场化的声音影响着团队的时候，唯一能让团队清醒面对的，只有团队的目标，以及团队中务实的工作态度。

公司季度大奖不是颁给想拿奖的团队的，也不是颁给能拿奖的团队的，而是颁给真正合适拿到这个奖的团队的。合适与否，不仅仅取决于你现在正努力着的东西，例如业绩，又例如你现在所认为的那些唯一衡量标准。

合适与否，是另一个决策。

七

最后，作为组织中的一个个体，我们需要知道：并没有人会关注“你的价值”。

对于公司，他们关注的是你对企业目标的价值；对于团队，他们关注的是你对团队目标的价值……类似于此的，只有在确定的目标下，才有确定的价值观，才有确定的价值。因此，“你的目标”才是“你的价值”的考核依据。而“你的目标”多数情况下是别人不关心的，所以也不要期望别人看到你的价值。

你应该感激那些关注“你的价值”的公司，这是不可多得的。但你尽可以感激这些得之不易的东西（甚至这也可能是假象），却不必期望可能得到它。因为，大多数情况下，HR（人力资源官）来找你只是因为你入职、调薪或离职。

他们并不会问问“你的目标”以及“你期望的价值”。

如果真的问了，他们也只是希望得到一个标准答案：你的目标就是实现公司的价值。

这是现实。

八

没有人是傻子，爱公司是一个玩笑。

因为公司是一个组织，项目是一件事。只有团队是一群人——我们可以爱这群人，爱这个团队；我们也可以在团队中找到归属感，找到自己的位置，找到实现自我价值的方法、途径，并为此付出努力。然而，大多数团队的可笑之处在于：如果它认为“你在为自己努力”，那么它会鄙夷你的利己。因为我们的教育总是在说：要为了公司、为了产品、为了团队、为了项目……

所以他们要么鄙夷你“为了自己”，要么无视你这样做。

其实，用人无过于四种境界，其一，我以己为器，认为一己之力便足够了，别人不堪一用；其二，我以人为器，就是把别人当成工具来用；其三，人以己为器，就是别人愿意把自己当成工具，粉身以报；其四，人以我为器，就是别人觉得做什么事都是在为他自己，而最终成就了你。

大多数个人，只是以己为器；大多数公司，只是以人为器。仅此而已。

这也是现实。

九

团队的核心是人，项目的核心是事，产品的核心是物。三者是企业经营中的三个不同视角，不同的管理者从不同的视角入手，所持有的观点、采取的做法以及关心的重点也都不同。

工程是什么？工程是实现（上述三者中的以“事”为核心的）项目的一个可选手段。因为在本质上说，工程讨论的也是事，以及事的做法。所以传统工程，也包括其他领域中的工程实施所讨论的三要素，便是“（做事的）过程、方法与工具”。三种要素都集中反映了“所谓工程，就是通过‘工程化的法子’来实现一个具体的目标”这一事实。

“立一个项”是我们常见的对开始某个事务的一般性说法。但就某一件事而论，具体做这件事的方法是什么，才决定了这一方法是否可以工程化。如果一个方法原本就不能工程化——我们举例来说，它过于依赖于一人一技——那么非要讨论“这个项目是不是一个工程”，在本质上就不成立。所以，公司大可以为种种事务而建立项目，但事事强调工程却并不见得可行（例如事事强调过程、流程、方法、工具化等）。

我们讨论的“人、事、物”三个方面都可以存在管理行为。仅其中的项目管理这一视角出发就有许多种不同的学派(13)，它们各自的关注点是并不相同的(14)。这些不同的学派，抑或说是不同的管理风格，也是适应不同的管理对象的。例如，一个管理者偏向于人际关系的梳理，那么他可能擅长团队建设与组织管理，但不见得能在生产型的项目管理中干得顺手，而且，当把他放在一个偏向“做事”的项目/团队中去的时候，他往往会显得过于官僚和权变。

“基于软件工程化的项目管理”是我们在本书中的一个重要命题。就上面的讨论而言，这个命题总体是有许多背景限制的。例如：

（1）是否有面向产品的组织结构；

（2）是否是做事的团队；

（3）项目对工程化是否有着本质的需要

等等。最后，管理者在项目的不同阶段是否有适当的管理风格，例如将自己设定为 Manager、Owner 或 Leader 等角色，也是“做事的环境”所讨论的因素。但在这个问题上，既可以通过任人治事来解决，也可以作为对管理者自身素质的要求。

————————————————————————

(1) 所以这是一个普遍性而非道德性的问题，切勿站在道德的至高点上来给出答案。我们可以要求一个力士做圣人，但如果所有力士都超凡入圣了，也就没有“圣人”了。

(2) 马斯洛是从人的“自我需求”出发来讨论的，因此所谓“利他”的结果必是“利己”的一种附加收益。

(3) 在创业公司，这可能是第二位的，大家勒紧了裤腰带过日子是因为大家都接受了“创业”这样的原始目标，对这一背景的“认可”决定了人的价值取向。

(4) 这也是本编的关键命题：不讨论用人术，不讨论成功学，也不讨论办公室政治。我们讨论的基点是“一个做事的组织”，这才是团队的根本，也才是一个项目所能依赖的基础。

(5) 本书的第四篇会更加详细地说明这一事实，并探究层次系统隔离“可变性”的本质。

(6) 例如，正是因为开发过程是测试驱动的，所以可以将测试视为“catch (…)”，而将“catch(){…} finally {…}”视为包含于一个迭代中的补偿。又例如，正是因为系统是层次化的，因此“try … ”才是有确定范围的，且“catch (…)”的捕获才是有效的。

(7) 这里采用了一个较特别的翻译。Owner 在词义上有“所有者”含义，一般译为属主、物主来表明这种关系。但在职场中，Owner 通常用于指代其责权，而不是确指职务，例如“这件事的 Owner 是谁”。

(8) 但对于许多人——尤其对于“官”字当头的管理者来说，第一反应往往是：“（这件事）我管”。我管与我负责体现了两种不同的角色定位，也决定了二者用以实现管理价值的方法的不同。

(9) 参考《经济学家茶座》2002 年第 3 期，“奥尔森学术思想介绍”一文，作者陈抗。主要观点与数据出自黎格曼的社会懈怠理论，以及奥尔森的理性自利的群体行为基础。

(10) 这是对软件产品定义与其需求类型的一个不完整考察，其主要目的是形成相互参照以便后续讨论。

(11) 其实并不是大家不懂得这个道理，只是大家觉得自己可以通过“种种努力”去影响决策，故而有了上下奔走，有了哭穷叫急，有了各种各样的组织活动。这些活动也的确增加了决策者在某些方面的砝码。但我的问题有两个方面，其一，这是否必需？其二，既使这一目的是必需的，那么除了这些方法之外，有没有达到目的的其他方法？

(12) 引自《南都娱乐周刊》2012 年 1 期，文《张艺谋：“我尽量保持平常心”》，采写记者谢晓、曾明辉。

(13) 参阅《项目管理技术》2006.10，以及 2007.01～11 期“项目管理学派”系列文章，作者 Rodney Turner，师冬平翻译。

(14) 泛义的项目管理对“项目是‘事’”这个观点是并不完全认同的。相反，上述这些学派将工程化（做事的具体方法）、团队（做事的组织）、产品（做事的结果与效果）等全部纳入了“项目管理”这个概念集中。举例来说，权变学派事实是将关注点放在了“人－组织”方面，而营销学派则可能将关注点放在“物－市场”方面，如此等等。

第二章

谋定后动：项目的存在权

势者，利之所趋也。

第一节　试错通常是无能的托辞

一

宋朝。福州。有富人跑去请教当时的知州孙觉，说他们打算捐几百万钱去修缮佛殿。孙觉是个慢性子的人，也就慢吞吞地问道：你说说，你们捐佛殿的目的是什么呢？大家回答道：求

福报嘛。

这个故事的上半部分是说：富人有钱，富人求福报。

接下来孙觉就说了：你们看那佛殿，坏也没怎么坏，破也没怎么破，修了它能有什么福报呢？话说回来，我们福州有不少的穷人，因欠了官府的银子而被关了监，受着牢狱之苦。若你们把钱捐给他们，便可以使几百人解开枷锁。你们说，这救人于难的福报岂不是更大？

这个故事的下半部分是说：穷人没钱，穷人得活命。

说来说去，这个事情好像跟孙觉没什么关系。然而当时，福州的监狱人满为患：行政开销是问题，国计民生是问题，官府形象也是问题。总的来说，这些是孙觉这个知州的问题。

孙觉此计一出，监狱空了不少。

二

看明白了这个故事的人，大概会说我不厚道，也或者他们会说孙觉不厚道：孙老先生算计来算计去，银子倒变成了官府的了！

眼里只有银子的人，眼界便太抵不过如此。

富人们要捐钱到佛殿，那便去庙堂里捐就好了，请问为啥非得跑去问知州呢？所以这个故事的底子里还有一层意思：富人在表面上是要捐钱给佛殿，以求得福报；面子下，却还是觉得自己这个行为很公义，很值得赞许，应当受官府的褒奖——好的话，给派个小官或任个会长之类的当当；即使差那么一些，也得立个牌坊授个匾额什么的。

面子上下的两个目的，哪一个才是真的目的呢？

我的不厚道在于说了一个权谋的故事，孙觉的不厚道在于看明白了富人们的用心。

做事情如果厚道是会受人赞许的，不厚道就会被人鄙夷。

当大家都有了道德目标，便没人考虑事情做得好不好了。

三

孙觉，字莘老。他与苏轼是好友，原本都是皇帝身边的重臣。莘老著《春秋经解》十五册，当时王安石一看便知道在自己之上。所以莘老其实是儒学大家，放在古代，就是在道德的至高点上了。这样的人的作为，我们（以当时而论）又怎能说他不厚道呢？

我们拿着现代人——或者仅仅说是现在的某些人——的价值标准，亦即“金钱物质的得失”，来评判孙莘老是行不通的。我们得放在当时的环境中去看，那样才能明白这个故事的有趣之处。

首先，于官而言，这些穷人犯了法而被收监，孙莘老就算体恤民情，也不能违法犯纪。表面上大家都知道得民心者得天下，但是把这些穷人都放了，民心看来是得了，法纪却乱了，天下还是要丢。所以孙莘老让富人捐钱，是在法律边界内解决掉“穷人坐监”这一问题的不错的主意。

其次，如果孙莘老逼着富人们捐钱，那富人也是“民”——所以到头来，看着是偏向穷人，却是害了富人，总的来说还是鱼肉了百姓。且不说富人们有没有路子去跟他打官司，单就情理上也是讲不通的。

再次，富人一方面要捐佛殿，另一方面又打着小算盘要求个功名。这功名原本就是“官”给的，给功名或不给功名都是官的份内之事。既然捐佛殿是求福报，捐百姓也是福报；既然求功名是求利，那便认可了这一行为，给些官府份内可给之利，又有何不可？

富人原本是不想把钱捐给官府的。但道理上，捐给官府和捐给佛殿的所得利益是一样的——更何况，就情势来说，如果富人们非要捐给佛殿，那就不一定能得官府的褒奖了。所以，既然所得利益一样甚至更多一些，富人把钱捐给官府也就合情合理了。

种瓜得豆是乐事，因为得瓜还是得豆是无所谓的事；事与愿违就并不可乐，因为当有某个特定的“意愿”时，事做得怎样就不打紧了。我记得 B 公司的一个故事：年终的时候，CEO 的业绩还算不错，但是大老板却把他撤了职。问之，大老板答曰：业绩是你们定的。

他要的是目标，亦即那种“特定的意愿”。

四

当然，这也要不违道德。

大老板是这样解释的：我们公司的价值观的第一条就是客户第一，你们做了一年两年，看起来业绩是有，但是我们的客户不满意，有那么多投诉，连我自己——作为客户去用你们的产品都不满意，那你们还怎么能说你们做得好？

所以一个 CEO 当然可能纵容下属去投机取巧——或是做好了业绩而做坏了行业，或是干脆失了道德去伪造了业绩的假象。然而放在一个道德问题上去炙烤的时候，这样的做法就是经不得考验的。至于大老板，他之所以忽略业绩，是因为业绩已经有了；之所以追问价值观，是因为价值观失了。

得与失，只是一瞬间的转变。这就是所谓形势。

势，就是利之所趋。看到利益是什么，看到利益的趋向是什么，也就是看到了势。所以孙莘老是看得到形势的人，大老板也是看得到形势的人。

大多数埋头做事的人，都忘了看看形势。

五

形势是两个东西，我常说“看山见形，看水见势”。在 B 公司的故事里，一时的业绩便是那可以看见的“山之形貌”，公司内在的追求便是那难于明辨的“水之势向”。

作为一个项目经理，所做产品便是确定的“形”；采用软件工程来做，便是具体的“法子”；而所谓客户需求，便是基本的“利益”；更进一步，用户的利益取向就是这个事情的“趋势”。

在一个产品开始做的时候，用户问的是“能不能用”，所以只要功能满足，“多快好省”的东西大抵都是满足用户要求的。这种情况下，快点做产品，快点圈用户，快点树口碑，都是当务之急，因为这些都是该阶段下用户所追求的。但到了往后一些的日子，用户积累起来了，需求分支变得混乱而复杂了，系统整体有了性能瓶颈了，这个时候再强调“快做快上”就事与愿违了——因为用户开始讲“好不好用”的问题了。再到了第三个阶段，基本的市场格局已定，用户已经达到了相当的基数，而竞争对手的关系以及领域细分变得微妙起来，这个时候再讨论“某一个用户能不能用或好不好用”的问题，便是吹毛求疵的事情——这种情况下要讨论的，通常会是领域扩张的问题以及新领域中用户的需求问题。

然而在第三个阶段中还有两个有趣的问题。第一个有趣的问题，我们既然说“用户的利益取向”是势，那为什么我们看起来却是忽视了“当前领域中的”用户的利益取向呢？

这取决于“产品的用户群体”这一定义是否扩大了。如果没有扩大，那么我们必以这个产品的当前领域为利益核心；如果扩大了，我们就得将整个领域作为利益核心。以 Adobe 为例，Photoshop 是为平面工作室而定义的产品，在这么多年的发展中从未改变过；与此同时，Adobe 通过产品线定义，将 Premiere、Flash Catalyst 等放在 Creative Suite 套件之中，来完成以整个“图形、图像艺术设计师”为对象的产品设定。与此相对应的，ACDsee 早期的产品定位为“阅图者”（viewer），在后期却扩大到“阅图者+制图者+数字图片管理者”这样一个复杂的用户群，并且仍旧使用“单一产品”这一概念来应对用户需求，以至于 ACDsee 5.0 Powerpack/Suite 之后的产品一直都无法满足各类用户的“集体意愿”。

形势在变，观念不变是不行的。一方面，单个用户既需要单一产品的便捷，又需要组合产品的强大功能；另一方面，企业用户既存在特定业务为核心的产品需求，又存在以部门与角色分割的功能选择。因此我们要么拆散了用户群，以不同的产品去面对不同的用户；要么以（可以自由组合的）套件的形式来满足用户对产品定制与搭配的需求。

我们自己也是用户。

六

第二个有趣的问题是，在这个阶段中，竞争对手可能还处于一个较小的规模之下，这时他们便会强调“能用、好用”的问题，而且还会以此为优势来打击你的产品。于是便会形成一种局面：大公司的东西“似乎”越来越不好用。

产品与产品线总会走入这样的困局。用户的口味越来越趋向于差异化，但产品要规模化生产与供应，却只能趋向于同质。这才是困局背后的根本原因。所以，一个工业化产品无论如何

都只能在“一定程度上”保证品质，而很难超越这个边界来使它变得跟一个小规模生产的（如手工生产的）产品有同样的性质(1)，或保证会受到某些特指用户的欢迎。

大老板是特指用户——注意，任何一个个体都首先是特指用户，这一点非常重要。正因为大老板也确知自己必然难于站在“典型用户”的角度来对产品性质提出意见和建议，所以大老板通常不会对产品的性质提出质疑。但是如果这一点能被确实——我的意思是说，某个管理者、决策者或管理与决策层能确实将自己放在典型用户的角度上，那么他们也只会在两个时机指出“产品性质问题”。

□ 第一个时机是管理层的变化。因为事实上产品性质必然存在问题，至少（我举个例子来说）在一段时间中总会有 5%上下的用户对产品持续不满，因此任何时间追问这个问题都必然是管理层权利变更中的一个权谋之术。

□ 第二个时机是产品的转型。产品转型并不一定是“变成另一个产品”，也可能是在产品面对的用户或面对的用户的品味方面的调整，因此当大老板“放大”了问题，则必然是试图通过对这一问题的锁定，来将产品调整到一个新方向。

如果大老板“不择时机”地、随意地对产品性质提出问题，那么他不但浪费了自己的一个管理工具，而且还使得管理、经营与决策三者在产品问题上辗转碾压、互为制约，进而产生大量的内耗。

“大公司的东西”总会对某一部分用户不好用，知道对某些用户以及某些需求说“No”是明智的，但学会“如何说 No”则是一门艺术。例如，公司可以宣布“我们只做高端消费者市场”，或者在招牌旁边贴上“五星级酒店”的标志，那么本质上就是在对某些普通用户说“No”。这种情况下，当某一个用户走进店内，面对种种价格乍舌的时候，只需要有服务生走近他，无比诚恳地说：先生，请问我有什么能帮到你的吗？

当然没有。事实上作为用户的我们总在被拒绝，只是因为被拒绝的形式不同，故而我们的反应也就有不同罢了。在软件产品上与此是类似的。“承诺产品特性”与“拒绝用户需求”是我们面对市场的两张王牌，但大多数的市场人员、产品设计者与经营者只懂得运用前者。这也就是 Acdsee 产品特性爆炸与麦当劳千店一味这两种风格在产品设计与经营理念上的本质差异。当然，这并不是说 Acdsee 就该“放弃产品特性”，而是说，Acdsee 在“做什么”这个问题上或许是对的，但在“怎么做”这个问题中却确凿无疑地错了。

七

“更多的产品特性”并不一定是正确的事情，所以看起来人山人海的开发活动中，各个小组在做的事情也就不一定“都”那么正确。总有团队、总有项目、总有产品会是牺牲品。切近形势的大体上会好一些、保险一些，但一旦形势逆转，又福祸两不知了。这也就是我们所谓的“大……规模开发”中的现实状况——所有人都在战战兢兢，为项目的生死而担忧，为“明天做什么”以及“我能不能做得了”而焦虑。

我在“大规模开发”中间加上了省略号来表示“无限”。因为在我看来，这是一种由“产品多寡”带来的虚假的繁荣。这种虚假繁荣背后是有理论支撑的，亦即“快速试错”与“短平

快项目”。前者是一种产品策略，后者是应对产品策略的项目方法——与此还多少有一定关系的，是敏捷开发方法。“快速试错”真的是可行的产品策略吗？答案是肯定的，但必须同时明确：所谓试错总是要付出代价的。这种代价要么是时间，要么就是组织。“较为明智的”做法大体上是牺牲二者中最小的一个。例如创业团队，组织规模是牺牲不起的——总的来说就三五个人十来条枪，所以就只好牺牲时间，于是创业迟迟不成通常是因为产品投入得不对，最后试错成本越来越高以至于不得不放弃(2)。又如大公司的产品线，在组织规模上“通常”是牺牲得起的，因而往往试图同时开发多个产品来及时响应不同的市场需求，于是组织规模越来越庞大复杂，最终组织成本大于产品成本，又变成得不偿失。

试错的定义终是基于成功二字的——成王败寇就是这么个意义。但是对于我们要讨论的对象来说，试错的对象不同，成功的意义也就不同。在团队、项目与产品这三个企业经营的关键视角中：

□ 如果试错以团队为目标，则显而易见地变成了组织重构这样的事情；又若以团队为试错基础，即团队可以自行选择要做的事情以及其价值判断，则是各自为战的战术方法。

□ 又如果试错以项目为目标，则容易多劳而无功，渐成组织内耗，例如，一个产品开发完就被扔掉便可能是运营部门、生产部门所谈的“项目”在内容上互不相同；但若以项目为试错基础呢，则项目是否作为基础单元以包含对所涉及全部资源的把控，便成必然要求。

□ 因此，第三个视角便成了更为常见的选择，即以产品为试错对象。这既在于产品的成败有明确的定义，又在于“面向产品的项目”的管理与控制过程在企业的组织形式上已经较为成熟。

总的来说，在这些视角下存在着职能部门、事业部与产品线这些不同的组织形态，以及各自不同的关于成败与价值取向的定义(3)。

试错的可行总的来说取决于两个方面的前提。第一个方面的前提是组织的可行性。在一个不适宜的组织结构上“试错”，其代价必将是惨烈的——所谓船大难掉头，倒不真的是“船太大”，而可能是船上的拖网太多，要先砍掉拖网才能掉头。所以，这些试错产品便是随时要砍掉的拖网，于是这些拉着拖网的人，便随时可能因为碍事而被推下水。进而，即使船再大，船上的人也都无不战战兢兢了。

一个组织，例如公司或部门，对于试错的结果是否有预期？对于试错的“错”是否能承受？对于试错的收益如何评价？对于试错团队的牺牲持何种态度？这些都是组织层面要考虑的事情。例如，如果我们要有尖刀班尖刀连，那就得有让他们“虽死尤荣”的体制与背景(4)。此外，“短平快”带来的通常是项目主体完整而缺乏细节。因而如何将一个“缺乏细节”的产品推到用户面前而不会招致反感，是一个组织化的行为，也就是说，需要市场、营销与产品的配合。因此，“短平快”如果缺乏组织力，与“死得快”是没区别的，开发完就扔掉、还没上线就说 No、用户三分钟热度等这些问题，基本上都是由此导致的。这也是大公司通常会做“小规模内测”，以及划分高端用户、体验用户、概念用户等特殊用户群体的原因——当我们把“试错”当成一种稀缺资源时，试错就变成了“试鲜”，成为用户有品味的、有 VIP 身份的或者有特权的一种表征了。

第二个方面的前提是试错的必要性。如果一个组织中的大多数项目都是在试错，这既对组织整体的积极性、荣誉感是一种打击，对经营收益来说也是巨大的负担。所以总体趋势上，我们是要减少（而绝不是增多）试错项目的。因此试错绝非“正常的产品决策”活动中的首选手段，它只有在当产品——必须再次强调，这里讨论的“试错”是以产品为目标的——面临类似如下无法直接抵达用户或抵达成本过高的困境时，才适宜作为一种产品的阶段性决策工具来使用：

（1）无法获得用户反馈来支撑它的价值（例如产品线中规划的关键产品环节）；

（2）无法与产品的直接用户沟通需求（例如通过市场调查报告来决策的产品）；

（3）难于获得用户使用产品的细节等。

永远不要试图生产一个“不知道用户是谁”的产品来试错，这绝对是系统性的灾难。反之，若是已经明确了或可以通过用户接触来逐渐明确了用户是谁，要什么，用什么，怎么用等问题，那么这样的产品就不是“试错”，而是“试对”了(5)。

真正有效的、可实施的大规模开发，既是组织上适宜的，也是方向上适宜的。在组织与方向不匹配的情况下的“规模庞大”，既是假象也是负担。而在这其中，“敏捷团队+测试驱动”只是适宜作为一种试错的方法而已(6)。然而无论投入产品的数量，还是实施方法的合理性，实质上都是无助于解决上述问题的。之所以我常常不讨论方法问题，而偏向于讨论方法的前提，其原因就在于此。我们往往在“技术上”准备好了试错，但试错的前提（组织的与方向的，可行性与必要性的）却常常不具备，这种情况下，看起来不错的项目或产品，其最终的失利自是必然，而其成功也多是机遇罢了。

八

做公司总是要挣钱的，不挣钱大家就得散伙；但是如果把挣钱这件事情说得那么直白，一定程度上来说，大家还是得散伙。这个问题我们在上一小节讲到过，我们如果鼓励公司以“经济价值”为先，事实上也是在鼓励“经济价值”为先的道德模式；如果以此为核心，则员工的利己行为便倾向于对“经济价值”的考量，倾向于“如何投入更少而获得更多”。

一种可能的做法是“在商言商，在民言民”。也就是说，道德标准对内和对外是不一样的，是一种选择性结果。例如，我们的市场人员去跟客户代表谈时会说：你看，我们做企业的总得挣钱，对吧？而对组织内却强调：我们都是一家人，所以应该互帮互助，对吧？

对吧？

其实，不怎么对。因为这是做人的做法，不是做事的做法。仅做事而论，如果是去跟客户谈判，那自然是如下这样的目标：第一，为公司争取最大利益；第二，为公司争取最大的合作空间；第三，在为公司争取最大利益与合作空间的基础上，让客户最大程度地满意……至于这样的目标，你告诉或不告诉客户都是不要紧的，因为他们必然知道。当然，如果你收受客户的好处，那也就相当于告诉客户你并不看重公司的利益与目标了。

再讨论“对内”的问题。无论如何，事该怎么做、该谁做都是和职业角色与责任相关的，因此第一等的话题是：

（1）该，或不该你做。

如果舍了这层关系，例如在创业团队中，大家都是有着共同的目标与期望的，也有着自己的能力与特长，因此第一等的话题是：

（2）我能做，我立即动手去做。

再舍掉这层关系，例如在一个松散的组织结构里，可能还没有既定的责权关系，那么第一等的话题就应该是：

（3）如果我合适做 Leader 或 Owner，那就让我来做。

这三种做事的法子（或态度），可以一言以蔽之：或尽职尽责，或各展所长，或当仁不让。

所以，所谓做事，也就是个能力与才干的问题。做事但求有尽心尽力的态度，这样一来，即使是看起来圆滑世故一些，也是受人尊重和堪能任用的。所以要鼓励员工做事，要形成“做事”的形势，那么公司也得是一个任贤用能、实事求是的公司。尽管从组织结构上来说，人的考察、调度与任用等，不免会涉及“做人”的问题，但这仅限于组织层面上就好了——我说过，管理者的利益得失要从组织层面上去找，也就是这个意思。

“先做人，后做事”是一种道德要求，而并不是做事的“法子”或前提。以做人论，宋朝的丞相宋庠(7)就曾经开了一个不太好的先例。他任守洛阳的时候，有一个仆人来检举主人行李中有漏税的东西。对于这件事，宋庠认为主人故然有错，罚缴税款即可；仆人控告主人，也是过错，所以打了一顿板子并赶了出去。在这个例子中，宋庠就是把法理判案这样的“做事”，套上了“做人”这样的道德规矩(8)。

结果呢？规章制度乱了，事情也就渐渐地乱掉。

人做成了好人，事做成了坏事。

九

当以道德准则来看一件事时，事做得好与不好是不要紧的；当以宏观趋势来看一件事时，事做得好与不好也是不要紧的。所以，做事的评价，不在于事做得好与坏。

而在于正确与不正确。

事可以做得漂亮，例如工程化可以细致到程序代码的行数统计，又例如流程的规定可以设计到一个人的左右手的不同分工等。总之，我们可以把一件事做成无可挑剔的范本，但是放到一个观察者的角度上去看时，它可能仍然是一件坏事。

我们多有做事的激情，而少有观察者的冷静。这让我想起在中式教育中的一个非常有趣的观念，叫“有眼色”。小的时候，我的父母亲便会教育道：家里来客人了要倒茶，大人在午休的时候不要吵等等，诸如此类。总的来说，有眼色的意思大体就是“知道在什么时候、什么情况下该做什么事”。

察情、知时、应势，那是很好的品质啊！哪个管理者、哪个公司不希望自己手边都是“眼里有活儿”的员工呢？然而为什么“有眼色”现在变成了乖巧狡佞的代名词了呢？进一步讲，我们的教育只强调培养埋头读书的孩子，以及埋头做事的员工。我们一面叹息大家眼里没活儿，另一面又把员工的脖子摁得死死的，不教他们有举头四顾的一丝丝机会。

眼色这个东西，用在人情便是察己识人，用在析理便是真知灼见，用在决策便是远见卓识，用在奉承便是奸滑谄媚……但是，这些都只是用的问题，而并不是“有眼色”本身的问题啊。做事与观察做事是两个角度。于前者，努力是必需的，认真是必需的，细致也是必需的……但这些都是“做事的必需”。如果你是管理者，那么你正好应该是站在后者的角度——观察做事。

所以你的评价依据以及时常反省的，应该是这件事的正确性。

第二节　合法的山大王为什么没能成功

一

我们谈到过：团队中积极或可能积极的成员约占到 2/3，而约 1/3（20%～30%）是“理性利己主义者”。如果单单看到这样两笔模糊的数据，那么团队大概是没有希望的了。

在大多数不太成功的团队中，2/3 的成员挣扎于这种“失去希望”的境地。他们或努力地避免自己沦为那些理性利己主义者中的一员（这事实上并不容易），或对这些人的存在视而不见。根本上来说，这是因为，对于组织整体而言，他们既没有权利也没有责任来处理这些人带来的团队问题。

这是谁的权利与责任呢？

二

我喜欢开会。这是很少人知道的一件事情，因为大多数同事都将我视为“会议敌视者”。事实上他们并不知道，在数年之前，当我的工作中有着大量的管理事务时，我总会“快速”地组织起一场会议。

——然后又快速地结束它。

之所以我们大多数人变得“敌视会议”，是因为那些会议组织者的无能，而非我们对会议这种形式怀有敌意——只可惜大多数“喜欢开会”的人并不能意识到这一点。他们的喜欢开会总是体现在：

□ 准备好记录工具，例如记事本或录音笔；

□ 准备好食物饮料，如果有两颗老年健身球之类的就更好了；

□ 找个相当适合开会的地方，例如茶楼或会务中心；

□ 让会议开始于自我介绍，而不是对问题的陈述；

□ 不设定讨论的范围，而寄期望于会议中“产生”点儿什么出来；

□ ……

而我则一直在努力地理解为什么我会敌视这一类的会议。后来的后来，我终于知道(9)：

希望自己独立解决问题的人，与希望延长解决问题的会议的那些人之间存在着矛盾。

三

“独立解决问题”是一种很特别的能力，它既决定了你在团队中的位置，也决定了你的组织风格——例如开会。

我能快速地组织并结束一场会议的基本原因在于，会议对我来说，主要是起到两个作用：其一，能快速统一一群人的行为；其二，能快速得到一群人反馈。我组织的大多数会议是发布决议或（统一地）调整工作方法、步骤与组织责权——简而言之，就是要让大家去做，或要让大家知道的事。例如，我可以为了让大家写一份好的 Word 文档而组织一次学习，但不愿意为讨论“要不要统一文档格式”而开一次会议。

我绝不会为一个会议而“设立一个问题”。这个问题若提出了，我必是在得到解的时候再开会(10)；若它是在会议中提出的，我会质问“为什么不在会前提出”并将这个问题直接纳入会后讨论。我并不习惯“用会议来解决问题”，或“在会议中解决问题”。然而，温伯格提到的“希望延长解决问题的会议的那些人”正与此相反，他们开会的目的就是为了“发现问题”，以及“部分或全部地解决问题”。

习惯于“独立解决问题”是领导者的一种基本素质。这也是那 66%的团队成员从众而你出众的基础条件之一。需要说明的是，在思维法上的“解决问题”是关注于问题的求解方法，而不强调它的实施的——你应当努力找到问题的解，而通过团队力量（而非独自）去实施它。然而，66%的从众者要么并不能发现这些问题，要么发现了问题而不主动求解，要么有了求解的思路而无法推动实施。

前两种情况是从众者本身“之所以从众”的本质特性，亦即他们性格构成的一部分；后一种情况则是组织的责权问题。

四

组织的“可笑”之处在于：并不是有能力发现问题、求解和推动实施的人，就一定是拥有相应责权的人。组织的授权过程是渐进的，他首先取决于组织决策者对你的信任，其次是你的能力，最后但也是最理想、最复杂的是二者的组合。

当团队陷入“从众+集体自利”的泥潭时，组织行为是唯一有效的解。当这种组织责权未出现明确授予时，就出现了两种情况：一是等待组织觉醒，二是团队内的利他角色的出现。基于奥尔森的观点，经济学家发现：不同文化的社会中都存在一些利他主义者——为数少于1%；模型模拟的结果显示，若这个比例提升到3%以上，则整个社会的风气和行为规范就会出现可喜的变化(11)。例如，三个和尚没水喝，但如果有人出来制定“轮流挑水”的规则，那情况就立即改善了。问题在于：

（1）“制定规则”这一责权是三个人中谁都不具备的，且这件事对这个人自身并没有任何特殊好处。所以他必是首先出于“有利于（包括其他人在内的、全体的）组织”的思想，才会着手制定与推动上述规则。

（2）这个人可能必须身先士卒地挑起第一担水，并且他又必然因此面临“从明天开始”没有人服从这一规则，而白挑了这担水的风险。

无论是谁来挑第一担水，整个的解决问题的方法都取决于“轮流挑水”制度是否能有效保障。而这只能依赖强权，或群众运动——这也可能演变为政治，例如争取一个从众者，形成2：1的局面以强行推动这一制度，并实施惩罚措施。

“轮流挑水”作为三个和尚问题的求解方案之一，将要求“规则制定者、维护者”这类角色的出现。这本质上是组织行为，且在现实中也通常是由公司通过组织委派来解决的。但作为一个有自省性的、有自发力量的团队，这个问题也可能被潜在的“利他角色”来解决——不过在大多数情况下，这个利他者将因为对责权的需求，而会表现得强权、强势和政治化(12)。

我们的教育中是反感这种角色的，但往往团队却渴望这种角色的出现。我们一面焦渴着没水喝，一面憎恶着团队的堕落，而又同时对那个跳起来试图解决问题的人投以白眼。渐渐地，一种坏味道漫散于整个团队：既愤愤于没有人解决问题，又愤愤于自己的想法不被重视，还愤愤于那个出头者抢占了自己的机会。

持尊重强者与不畏强权二者之一的，都是弱者；二者兼备的，才是强者。

五

领导者是那个能独立解决问题并拥有相应责权的人。一部分人正在成为领导者的过程之中，所以他们表现得聪明、果敢，富于决断，并且处于权力的上升期；另一部分人已经成为事实

领导者，因此在权力方面处于衡定期，而权利问题就变得明显起来。

“因权而谋利”并不是一件坏事。事的好与坏是有角度的，就如同我们前面举的美国民众的例子一样。权利——作为一种利益——是有视角与立场的，当为民众谋利时大家就都支持都赞成，而当大家发现你在谋取小集团利益时，民众就开始反对了。

群众的眼睛总是雪亮的。

这就回到了我们对问题的原始设定上：团队的普遍利益是什么？我提到过以下几点。

（1）项目利益：项目是面向“事”的，所以必然要求这是一个做事的团队，所以有事做、做对、做好是基本前提与利益核心。

（2）个体利益：这个团队中的成员是各有私利的，各角色有其在利益上的基本底线。

（3）团体利益：可以参考马斯洛“需求五层次”来考察团队整体的利益。

而领导者的眼光，或者说他对大势的主要判断，便在于他如何认识什么是真正应当寻求的“利益”。

六

那么我们来讨论一下“组织的革命”。

随着新任 CEO 的上任，C 公司面临着一次重大的内部组织变革。这次变革概括起来有如下几点：其一，从产品管理走向产品线管理，在全公司范围内，有生产（包括内外部的软件开发）活动的都称为一个产品线，名曰“山头”；其二，权力下放，山头设有“山大王”，其内部组织由山大王在全公司内选择，产品线内部决策可以山大王一言而决；其三，资源放开，山头经营所需资源作为预算，由山大王提出并报批，按预算计划拨给。

这次变革的核心动力来自于两个方面。其一，原有产品管理过于碎片化，资源使用上的冲突面临失控；其二，新的管理高层对于既有业务的体系和方向并不明确，寄希望于此次改革能产生新的团队、业务、产品线与经营模式。大体上来说，这样的一次变革涉及很多部门的重新组织，也涉及许多的产品与产品线的启动与中止。当然，不可避免地也涉及了许多的权力、资源与组织关系上的冲突——但这些，是在这一变革的决策中可预期并计划承担的代价。

但这样的一次变革的结果如何呢？

事实上这次变革除了在形式上的轰轰烈烈之外，其他的几乎什么也没有改变：

□ 原有产品一个没少，还产生了更多的产品，并因为产品线的阻隔而需要更多的资源；

□ 原有的大多数职能性、管理性的部门并没有撤销，并按着原定的规则继续要求评审、考核、组织会议和配给资源等；

□ 原有企业的经营目标被暂时性打破或置疑，但新的目标尚未明确，因此各个产品线处于“自我规划”的状态，完全缺乏通盘的考虑；

□ 由于资源存在竞争，因此在各个山头的规划中普遍出现了“过高地设定目标”的情况，这一方面可以使规划更有吸引力，另一方面也有利于争取更多的资源；

□ 原有的软件开发过程没有变化，仍然基于“产品需求文档”（PRD）并基本采用瀑布模型来推进工程；

□ 部分项目组尝试新的工程过程模型，却因为公司组织机构上既有流程规范上的限制，以及公共流程环节（例如产品部门）存在资源竞争而无法有效推进——通常情况下，为避免此类资源竞争而设定了计划、优先级与流程，而这正好又是体制性的；

□ ……

如上所述，整个组织与经营活动在短时间内乱象纷呈。但事实上管理高层并没有为之所动（这也是决策力中相当重要的部分），因此这一决策仍然被忠实地贯彻了下去。这些浮现于表面的问题点，既是一种战术需要，也被视为是战略推进的必然阶段。也就是说，如果乱象是暂时的，而其收益是长期有益、有效的，那么这些代价是可以接受的。

什么样的收益呢？

前面提到过这两项重要的收益。其一，是否可能在这个过程中产生新的经营模式？其二，是否可能顺利过渡到产品线管理？如果这两项目标达成，且这个“乱”的过程（在资源和时间上）是公司可以承受的，那么这将会是一场成功的改革。

但事实是：这次改革彻底地失败了。

七

看起来，这个策略本身存在很多的问题。例如，产品线过多，并将因此导致跨产品线的决策效率低下；又或者，产品线为了安全起见，完全可能在新产品的思路上趋于保守；又或者组织层面并没有解决资源竞争问题，因而增加了失败的风险……但这些都不是这次改革失败的核心原因。因为我们定义一个战略的失败与成功，并不是看细节上的代价与冲突，而是关注它在战略目标上的关键设定是否达到，或是否失去了达到关键设定的基本条件。

对于这次改革来说，其策略的关键点（即其基本条件）在于“产品线上是否有真正的领导角色”。这既可以通过组织任命来达到，也可以寄期望于在经营过程中有人能脱颖而出，或者还可以通过刻意地战术侧重来形成。但这三种情况（或更多种可能）都必然有明确的前提，也就是我们需要清楚地知道：一个合格的山大王应该是怎样的？

这个问题却正好在改革之初就没有明确。回顾这次改革，C 公司设定了“新的经营模式”与“产品线管理”这样两个目标，但却没有仔细观察在组织中是否真的有具备相应能力与素质

的管理者。也就是说，我们并不确知组织中是否有这样的人；又或者，即使我们先有了某种确定的组织，我们也不确定有人能胜任所需的组织角色。然而即便如此，这一革命性的战略仍然是相当值得尝试的。如果我们确知“我们要找的人是什么样子的”，那我们可以先改变组织，以便等着那样的人“跳将出来”——尽管这已经是接近太公钓鱼似的妄念了，但它毕竟仍然是一种可能的选择；而且事实上在创业团队中，自发的角色产生仍然是一种主要的模式。

如果我们真的要做这种尝试，那么我们究竟要以什么样的“山大王”为标准呢？按毛泽东的说法就是：就算当山大王，也要当“有主义、有政策、有办法、闹革命”的山大王(13)。

首先是“有办法”。这是个方法问题，如果一个人在山头中是明确的探索者、实践者，有着层出不穷的“鬼点子”并努力付诸实践，那么他便有这样的基础条件。大多数从众者愿意跟着“有办法”的人走，这是简单的趋利模式。也就是说，看起有“有办法”的人胜率总是要高些。“有办法”是一个团队的基础条件，但不是领导素质，也非领袖气质。举例来说，团队中可以有智囊，可以有军师，因而领导可以通过决策力来解决“办法”问题。

“有政策”通常很有可能是外部——更高层级的组织给予这个山头的限制，这更多地出现在既有的（而非新创的）组织结构中。例如，红军的“不拿群众一针一线”就是一条很有效的政策，它约束了山大王的行为，是红军之所以成为革命之师，而不会变成军阀流寇的政策保障(14)。另一方面，“有政策”也是对这个山头经营方法的细化和规则化，是对所谓“有办法”的一个补充、概括和提炼。领导者对于方法有着天生的敏感，但又非学问家似的书面化，他必以简洁可行的方法来形成可实践的、可集团化的规则，这是“有方法”之于团队的价值。而这一点，也是 C 公司的“形成新的经营模式”这一战略目标所需求的高度概括能力。

战术与其规则化，都是可以学来的。比如，当时井岗山上的山大王、土匪朱孔阳的战术就是“不要会打仗，只要会打圈”——只要部队还在，就可以打仗，就还可以继续做山大王，所以朱孔阳的办法归纳起来还就是“山头策略”。而毛泽东的高明处之一在于将“打得赢就打，打不赢就走”、“必要时拖队伍上山”(15)等类似山头策略的办法，变成了著名的《论游击战》。但是，远远不仅限于此，毛泽东非常明确地指出(16)：“游击战争一刻不能离开民众，这是最基本的原则。”游击战并不是简单的打打跑跑，它在指导思想上就被毛泽东分解成了“游”和“击”二字：“打开以发动群众，收拢以应付敌人。”这样一来，当“游”是有目的地发动群众时，“游”就不是东躲西藏了；当“击”是收拢而战时，“击“就表现为以强击弱了——游击战整体既是有目的的“游”，又是有目标的“击”。

这样的战略、战术与战法，是有系统性的。这既有系统的关系，又有系统的背景，这一背景即是游击战的基础：只有在“闹革命”的地方，游击战才有战的必要性；只有在“有主义”的地方，民众才能配合共产党打好游击。“闹革命”与“有主义”既互成方法，亦互为思想，还互作背景。这事实上也区别了“山大王”与“领袖”的不同。毛泽东等老一辈革命家自言上井岗是当上了“山大王”，但他们走得进井岗山，也出得了井岗山。从进去到出来，他们自始至终都是“有主义”与“闹革命”的，这就是他们与历史上众多的山大王不同的地方。

纵观整个反围剿与抗战，共产党一直有战，一直有胜，也就一直有着最终胜利的希望。而你坐镇一方山头，到底要给团队什么，才能让团队跟着你走，或按着你的路线走？看来做个山大王也不是容易的事。这在本质上是对领导角色的要求，即方向明确，制度有效，知道怎么

做，知道做什么。

八

有合法的山大王却仍然“群龙无首”，这就是领导职缺位的结果。有了“职”而不确定其“务”，不确定其“权”，不确定其所谋之“利”，这个领导的所有作为都将失去支撑。反过来，势者，利之所趋也！我们明确地知道：

由个体利益、团队利益与项目利益所构成的大势是项目的整个背景。

而一个有团队价值、为项目负责的领导者，其核心在于：如何正确认识与统一这些利益？

“合法的山大王”们，无论是因为何种原因得了山头，也无论他们在具体的行事当中是如何地“有方法与有政策”，若失了行事的方向（闹革命）与成事的背景（有主义），那便也只不过是闹腾一时的山大王。C 公司在这次变革中的策略不见得不好，但这一策略的两个主要背景，亦即形势，亦即基于形势的主要判断——行事的方向与成事的背景都并不清晰。前者，亦即是行事的方向的不明确，导致产品线划了等于没划，原来试图通过产品线来归拢并进一步削减产品，却因为方向不明确而“不知道怎么划”，削减也就无从谈起；进一步地，产品线的做法与产品的做法没有不同，新的流程与方法又受限于旧的体制与流程，体与制互成掣肘；再进一步，各个产品线又“貌似”获得了授权，故而时时有“自作的主张”而无视（也无法视见到）全局的利益。后者，亦即成事的背景的不明朗，使得“形成新模式”这一构想变成了赌博，寄期望于十几个山头的相互竞争中会有异军突起。

果然，这些产品线折腾了一年之后，终于有一条产品线“似乎”杀出了血路。而当大家纷纷看齐，准备学学他们的“方法与制度”时，这条产品线却一夜之间崩溃了：在业绩上存在着不道德的竞争行为。大家这才发现，这些山头事实上并没有出现过真正的“产品线”，那些貌似“产品线”和“新模式”的结果正是虚假繁荣的一种。这才恍然大悟，缺乏一个真正的领导者，缺乏领导职能中最重要的“方向设定”与“背景判断”，团队的成功多只是暂时的现象，或某些特殊因素导致的表象。

如同我们开始讲到的，当团队处于这种困境中的时候，才是彰显一个企业的组织力的时机。若“找到合格的山大王”是关键点与基本条件，那么这场变革在三个月内就可以结束了；然而在山头立起来了之后，却不太可能三个月就削平。这样的兵行险招儿，是在我们无法对组织背景进行精确判读时最选无可选的策略。

真要这样做，也总得有人守在岸边看着，别让兵掉到水里。

第三节　自己想办法

一

如果说马桶“是 20 世纪最伟大的发明”，那么粪坑便可以说是整个人类从愚昧走向文明的标志了。动物除了少数的几种会掩埋粪便的之外，实在没有谁会像人类这样在地上挖个坑来解决腹中污秽了。上面飞的鸟类会把人类的头顶当做粪池，下面跑的兽类会在地上设下“陷阱”，水中的鱼类则更简单、直接、正大光明的地污染着水源。

所以人类学会了挖粪坑。自从有了粪坑这个伟大的事物，原始人类才开始摆脱通过气味来寻找居所的“本能行为”，从而深度刺激了脑部记忆功能的进化，脑容量随之增加。进而人们开始了共同利用粪坑的群居生活，有了原始部落，以及群体、社会等组织形态。

粪坑对于原始人的重要性远比今天的马桶重要。马桶隔离了人们之间的交往，各家专用的马桶使得人们的沟通日益艰难，甚至成为奢望。而远古的人们与马桶时代之前的人们一样，可以在两个粪坑之间，在一道小小的、类似于篱笆的障碍物之间语言交流、学习和沟通，甚至可以互通有无，解决类似于没有厕纸这样的尴尬小事。

说到厕纸，这也是另一个令我充满敬畏的发明。因为自从看过《超级战警》(17)这部科幻电影之后，我就一直被一个问题深深地困扰：如果没有了厕纸，该怎么办？在这部科幻片中，到 2032 年，那时的厕所里就不再有厕纸了。

在相同的位置上，放着三块贝壳。

原始人可能从粪坑上传递木棍这样的事件中，理解了互助与友谊。而马桶则逐渐地加深了人与人之间的防范与私隐。马桶与厕纸带来了文明的如厕方法，而我们却再也无法让自己回归原始：贝壳与木棍，只是材质的不同。

在 2032 年的未来，什么才是可行的如厕方法？

二

古人是很讲程序、方法与规则的，这其中就包括如厕这件事。而且，这件事的程序性相当重要，器具制式和用法也有一定之规，以至于需要佛祖释迦牟尼来详加解释。这件事情记在《毗尼母经》第六卷里，要求如厕者常自备“厕筹（木片）”；如果没有，那也不能拭在墙上、厕板上，也不能用石头、青草、土块软木之类代替；而应该现找木、竹、苇，临时用作厕筹。不单单如此，佛祖还解释了厕筹的长短制式，以及用完了之后不能用弹、振、甩这样的法子来弄干净，也不能跟干净的厕筹混放在一起。这个，便是佛祖他老人家传下来的“上厕用厕筹法”。

佛经原文大抵便是如此。所以那个时代的僧侣们，大概是要随身备厕筹一枚的。倘若一时忘掉，那便只能折了木、竹、苇来替用。再如果这些东西都没有，可能就必须蹲在原地，等着其他人来，或是借用，或是求人家去取。但是，顺着这个思路想下去，我便发现：如果僧侣在四野无人的地方遇上了这个问题，既没有人又没有替代物，那么岂不是要活活地“蹲到死”？

几千年了，从没听说过有佛教徒这样蹲到死的。即便这个范围放得宽大一些，我也没听说过有哪个人是会这样蹲到死的。

没有人会机械地遵循某种“如厕方法”所约定的过程，一旦有“标准方法”所未顾及的、不太周全的情况发生，我们总是自己想办法，去把这些问题解决掉。我从来没听过会有人自己提着裤子出来，备好了厕筹再蹲回去完成程序的。

而在我们的工程中，“过程论”的专家总会要求你这么做。同样的，这样的专家在面临三块贝壳时，宁可触墙而死也不会上厕所，因为没有任何可用的过程、方法与工具。

三

佛祖其实是个好人，他老人家介绍自己的如厕最佳实践，包括厕筹最短也不要短过四指，要不用着不方便；又包括不能把染着污秽的厕筹甩来甩去，要不沾到别人，起了纠纷，打得头破血流；还包括用过的厕筹不应当乱放，要不传染疾病或是影响别人使用……这些方法的确是很好的，体现了丰富的如厕经验，并总结得条理清楚，可行性相当强。

但无论如何，你得先有厕筹。因此工具论也就大行其道，以至于南唐后主——就是写“春花秋月何时了”的那位了——也要亲自去削竹片木片来做，并且还要放在脸上试用，若有锐利不适的地方，再加以修正。所以厕筹这个东西，也是可以做得相当考究的，例如在外面拿香囊之类来包裹，既款款有型又不失雅趣，这样的事情在东晋也是有的。

先有了方法，而后其细节的步骤被规则化了，就变成了过程论的论调；其细节所依赖的那些东西，就构成了工具论的基础。如此一来，“过程+方法+工具”，就构成了我们将一件事工程化的全部。

当一件事情的做法被工程化之后，事就不再是具体的事了，成了一类工程。或是统一工程，或是敏捷工程，总之都失去了“具体”二字。

不再具体，也不再立时可用，也不再（仅仅）是一个法子。

四

我们原始的目的仅仅是“方便”而已，最终却纠结于“怎么擦干净”。

还是先拉完屎再说吧。

这件事得自己努力。

五

软件工程原本是只讨论“过程+方法+工具”的，这是纯粹工程学的视角(18)。这其中即使有一些看起来与人相关的话题，也不会被过深地讨论到。例如，过程中显然涉及人，但不会因此而讨论到“过程中的产品环节是否可以由开发人员来兼任”这样的问题。事实上，在“纯

粹的”软件工程的讨论中，过程的各个环节都是由一些“角色”来推动的，这些角色如何存在于一个具体组织的部门中，其职业定位、能力结构等都并不算一个严格意义上的工程话题。

但是这样的“纯粹”显然遇到了挑战。当我们试图将工程完全地理论化之后，它就必然面临实践的困境。“人”的问题在这方面首当其冲。《人月神话》并不是一个“人/月”的度量理论那样简单——事实上那本书很少讨论工程度量的问题。《人月神话》在工程上的重要性在于，它严肃地提出了“人，是不是工程学问题的一部分”这样的话题。这本书的答案将这个话题引向了一个极其巨大的迷局，即如果需要更多的人来实现工程，则应付由人带来的复杂性可能将超过工程本身的技术复杂性。《人月神话》一方面有先见地对技术复杂性提出了一些可靠、可行的方案，另一方面也悲观地认为由人带来的复杂性必然导致“巴比伦塔”最终的倒掉。“没有银弹”的论证过程将所有的焦点集中于：通过（较小规模的）程序实现的过程，无助于求解（包含大规模工程在内的、普遍含义的）根本任务。

作者 Brooks 把三个问题从《人月神话》的讨论中摒除了出去。其一，项目必是有始终的；其二，哪怕是一个不成功的产品，也是需要交付的；其三，团队是人的问题，并不是事的问题。《人月神话》将这三个问题的前题隐设为：项目需要承担大量的附加责任，产品必须是尽善尽美的，以及人是私利的。而这正体现了“工程学”的学术本质，即如果我们可以的话，应当基于工程来解决一切问题；因而它必须首先是能解决一切问题的。

在这些年的实践中，这是我所见的、对“一件事”具有最大伤害的但又貌似义正辞严的、跨越行业与组织的普遍观点了。我之所以用这样复杂的定语来说明这一“普遍观点”，是因为它确实那样“自然而然”地存在着，就像我们每个行业都受了同一个神灵的启迪一般。事实正是如此。任何一个有道德、正义而又富有职业责任感的产品经理都会跳出来，说“这个产品没做好是我们的责任”，然后历数种种设计细节，再加以细化，并立时加入你的需求库；任何一个有同样品质的项目经理也都会跳出来，说“这个项目没做好是因为我们对流程的贯彻不够”，然后开始制定更详细的管理措施；然后开发人员也跳出来，说出超过 200 个的技术改进点……

但每个人都将问题揽在自己头上，这难道不是一种好的品质吗？如果是，那么我又怎么会用一种调侃的语气来说出上面的假设？

这样的假设，在于警示我们一个被忘掉的事实：之所以这是一个系统，并不在于一人一已或者一个角色的过失，也不取决于某一个角色的单方面的努力。这些“积极、冲动而又充满道德责任感”的人，事实上是无助于找到系统问题的答案的。现实中，我们可以找到许许多多具有这种性质的工程、产品与过程模型。在这样的模型中，（提出模型者所代表的）中心角色总是认为自己应当能够并且也必须能够去承担更多，因而也就更重要，因而其他所有角色都应该围着自己打转转。

然而这样一来，每一个角色都站在一个自己认为“更合适”的角度来看我们在做的事情，而全然不管那件事情本身为何。例如，开发人员可能认为自己的某项技术发明有着相当“巨大”的前景，于是期望着有一个团队来配合自己将它实现为产品。至少在他看来，由技术推动产品和市场是相当伟大的成就。但是，一旦他提的这个想法如同扔进了泥坑一样丝毫得不到反馈，那么他又立即开始着手“发明”下一个技术与想法了。

你会发现，他原来根本不需要对产品和市场负上任何责任的！他只管提出，而不需要负责任，那么又怎么可能站在别的角色上去思考问题呢？

所以我再读《软件工程》这本教材的时候(19)，就只会看到一些“管理的技术”了。然而我从不指望，把“管理”当成技术手段和工程方法的人能真正地做好项目。

我的意思是那个具体的、眼前的项目。

六

如上所讨论的，在发现“人的问题”是一个关键因素之后，传统工程就将它作为一个学术话题，用一些角色、规格与方法来限制这些人的行为，试图将管理变成一个“技术所需的条件”以及“可以基于技术手段来改善的群体工件”——“人件”这个名词其实就翻译得很好。《人件》本质上也是基于这样的立论来讨论工程话题的一本书。简单地说就是：总有那么一些方法可以让管理成为工程的一部分。

但是敏捷工程却在“人的问题”上探索另一个“解”。关于敏捷工程，我从来都是把它与传统工程“对立”起来讨论的。但在根本事实上，它们是不对立的——我的意思是说，它与传统工程在“关注人的问题”这一点上并不对立，但在二者的求解方法上却大相径庭。

敏捷工程直接将问题指向“人的私利”与“合作”之间的矛盾，即人们“究竟是为什么要合作”的。敏捷工程无比强调团队内部的合作，并将这一合作设定为一种“道德需要”。由于有了这个前提，不合作者将被套上道德枷锁，或被拷问，或被驱逐。总而言之，敏捷团队必将因此变成一个（尽可能地）倾向合作的团队。因为这一核心前设的不同，在四条敏捷宣言中就有两条涉及合作（另外两条则是针对产品定义的），而在十二条敏捷原则中则有超过半数都是在强调团队成员的道德认可，而非强调某种技术方法。

从这个角度上来说，敏捷工程的成功之处在于形成益于合作的团队氛围，以及在道德与行为准则上实现“强制合作”。

七

无论敏捷工程还是传统工程，都是无法为你正在做的这件事提供直接答案的。

这件事得自己努力。

一个具体化的工程涉及三个方面的问题：人与团队，产品与用户，项目与方向。如果这些东西要被“系统化地”组织在一起，必然是需要将它视为“一件事”的，这也是它被“立项”的目的。如果这一前提不存在，则产品只是用户调研、产品设计或产品线上的一个规划，而团队则可以简单地视作“一群人”。只有要做一件事，这三个方面的东西才发生了关系，也才交织在一起成了“问题”。

要开始做“这件事”，会有不少的前提条件。例如，作为一个项目经理，你或许要求“先做三个月技术培训，再开始开发”。但事实上这些都算不上“前提”，因为你显然也可以不这么

做就匆匆上马——只要你有机会在团队行进中调整大家的步伐。唯一能作为“开始这件事”的充要条件的，事实上只有一个决策性的判断：我们是否需要用工程化的方法来实现一个产品目标。这里涉及下面三点细节。

（1）是软件产品吗？如果这不是一个需要“软件生产”的产品，则它可能不需要“软件开发”这一过程，固而也就不需要我们的软件工程。例如，市场部门需要了解客户反馈，那么它既可以实现为一个在线聊天的客服机器人，也可以实现为包含大量职员的客服部门。而后者显然是不需要“软件生产”的。

（2）需要开发吗？即使是一个软件产品，获得它的方法也可能不是“软件开发方法”。例如，上述的“在线聊天客服机器人”，如果我们将它作为一个产品外包，那么“外包”这件事对于我们来说就是一个供需管理，而不是“工程化”的软件开发活动。

（3）需要工程化方法吗？一个研发性软件产品的开发过程，可能不是“工程化”的。例如，我们真的要在公司内立项来做上面这个产品，但我们只分配了一个开发人员或一个临时小组，产出的产品也不用于市场销售，对时间的控制也可以放得很宽轻，而且最为重要的事情是，假定这个项目是基于技术研究的，因而当它失败了也不会有什么负担，那么我们便不需要考虑“是不是需要一个工程化的软件开发方法”。

当决策判断发生了，当“应作为一个工程化的软件产品来予以立项”成为事实的时候(20)，产品、团队与项目三者就必须立即成为“具有相同目标”的一个合作群体。这个相同目标，就是“做一件事”；这个相同目标中的道德原则也只有一个，就是“做完”。

我知道有一个项目，一个做了整整五年的项目。在项目坚持了这么久之后，是什么在支持着这个团队还保持着追求成功的激情？就这个问题，一个普通成员的回答典型、普遍而又发人深省：

“在我们的头脑里，完成一个项目是最最重要的事情。”另一位成员说：“即使我们不喜欢这里，我们也不会让项目流产，我们宁愿先完成项目，然后在第二天辞职。”

这个团队就是 Windows NT 4.0 的开发团队。在读《观止——微软创建 NT 和未来的夺命狂奔》这本书的过程中，这名普通的甚至没有在这本书中留下名字的成员，说出了让我如此印象深刻的话。这样的普通成员，正是这个团队整体的职业精神的缩影。

回顾这本《观止——微软创建 NT 和未来的夺命狂奔》，我们是无法孤立地站到团队、项目与产品三个视角之任一来思考的。从根本上，具体工程最重要的话题就是这三个视角必须是统一的、系统的以及面向具体的“一件事”的。

————————————————————————

(1) 这里说到的是“性质”，而非品质或质量。手工生产或精细生产并不是品质或质量的唯一保障条件。

(2) 很多人庆幸在这一过程中“获得了不少的学习经验”。但这其实已经失去了“创业”的初衷，是最无可奈何的下下限而已。

(3) 例如，职能部门以部门的、人的利益为核心，事业部以一类事务为方向，而产品线关心某类用户的需求等。

(4) 壮烈牺牲是与国家的烈士抚恤，牺牲、病故抚恤和伤残抚恤政策等一体的，例如民政部门就分管革命烈士褒扬，指导优抚事业的管理等事务。否则，烈则烈了，壮则无闻矣。

(5) 许多类似“成功学”的励志访谈中，创业者的经历总是被解说成充满机遇、运气与挑战的试错过程。但细细分析，大多并不是因为有不怕试错的好运气，而是他们往往在选择性地“试对”。

(6) 试错的基本保障在于成本可控，而这一模式在本质上有两点是适宜试错的：其一是团队规模，其二是开发方法。

(7) 音“祥”。

(8) 这个故事的背景是在讨论儒家的“尊卑有序，下不欺上”的道德观。这样的道德观本身带来的是类似“欺下谄上”的道德败局。但我们这里并不讨论道德观本身的对错，而是讨论治人与治事之间的冲突。

(9) 引自《成为技术领导者——解决问题的有机方法》，温伯格著。

(10) 例如，“要不要统一文档格式”，若它是一个问题，则我所组织的会议就是用于通告“统一，或不统一”这样的一个决议的。若是前者，则可能还包括一系列用于培训“统一格式”的学习课程，并且在开会之前，这些课程就已经定下了。为了达到这一目的，我可能已经走访了一些人、一些部门，并组织了培训讲师、制定了时间计划。

(11) 引自《经济学家茶座》2002 年第 3 期，“奥尔森学术思想介绍”一文，作者陈抗。

(12) 也许所有的和尚都知道“轮流挑水”能解决问题，但谁跳出来制定和维护这条规则谁就是众矢之的。真正的“领导者”敢于把自己置于焦点，并同时有具备让自己不被射杀或灼伤的能力——在这个过程中，“为谁谋利”是所有强权、强势和政治手段合法化的基础条件。例如，美国民众就向来是“国家在国际事务中应起到领导作用”的支持者。

(13) 引自《历史选择了毛泽东》，叶永烈著。

(14) 按照我的惯例，便在这里留一个彩蛋吧。试问：“不拿群众一针一线”到底概括了哪些管理方法以及对管理效果的评价？

(15) 引自《朱德选集》，文《在编写红一军团史座谈会上的讲话》（1944）。

(16) 引自《毛泽东文集》，文《论抗日游击战争的基本战术——袭击》（1938）。

(17) 英文片名 Demolition Man，马考•布莱姆比拉导演，西尔维斯特•史泰龙主演，1993 年上映。

(18) 事实上，还会讨论到“目标”问题。但在传统工程中的“工程目标”并不是确指，而是使用“需求”来指代的一系列技术方法。传统工程基于“需求”而非基于“具体目标”，并且——与此相关的——它与架构的“面向问题”也是互相背离的。因此传统工程中的工程师视角下的思考，对“架构师”这一角色在团队中的融入帮助并不大。

(19) 早些时候的《软件工程》教材是不讲团队、管理等内容的，但后来它们都逐渐地加入了相应内容。但老实说，这些书——例如《软件工程——实践者的研究方法》与《软件工程（Ian Sommerville 著）》等——中有关于人的讨论都是隔靴搔痒，无济于事的。这些经典的、学术化的工程方法仍是以自己为中心的，它们试图把管理学、组织学或社会学“拿来一用”，其思想离“看到一个项目自身的系统性”还相当之远。

(20) 事实上三种情况的反例都是可以“立项”的，譬如组织建设或人力规划的项目，企业资产购置与管理的项目，实验性项目。它们作为项目是满足“时间、质量与成本”这样的项目要素的设定的，但却都不是工程化的。

第三章

具体而微：工程是系统而不是事

我看到的一种可能的答案，就是“组织”。它远远地在 EHM 图的最外层——那个层面约束了工程、实施的形式，并适时调整着这种形式；那个层面看得到工程的目标方向，也适时地调整着这种目标方向——真正决定着工程生死的，在工程之外。

当然，工程本体的病患是有的，例如“牛屎图”中的工具与方法不匹配，或者质量模型中的时间与质量起冲突等。这些病患并不会因组织的调适而消亡，但是组织的调适能给出治疗这些病患所需的时间、空间和基础。所有的病患其实都是长在组织这个“体”之内外的，当组织环侧与开放自身，毛病就显露了出来。这如同人的屁股上长了疮，总是要抬抬屁股才能抹得上药的。

从这一点来看，“从局部看来，向全局看去”，既是我这些年来工程实践的所为，也是我将来必行的方向。就软件开发而言，跳出具体角色的限制，看到“一个组织”和“工程的实施”的全局，也许正是破局的机要？

第一节　做事的选择

一

大多数讨论组织及其管理的书籍总把一个问题谈不好，这就是“权”。中国人善权畏权，而又避免谈权，这是一种普遍的处世哲学。然而因此绕开了这个字去谈组织与管理，总是戳不到痛处！

上一章绕了很大的圈子来谈“因权而谋利”，本质上就是指出了“权”的一种出处，即由于它代表了一部分人的利益，因此得到了支持，而这种支持就变成了“权”。例如，共产党代表全民利益，其出处就在于毛泽东等老一辈革命家对“天下穷人要闹革命”的时局判断，这显然不是出自某种权力机构的授予。而“合法的山大王”则是第二种权的出处，即授权。尽管两者在由来上不同，但都表现为一种“权”。

C 公司中的“合法的山大王”最终没有取得成功，从根本上来说，是获得授权的管理者并没有做好准备。管理者并没有意识到，在组织变化时，最大的、最本质的变化产生于“权”的变更；或者，大家已经意识到了，却也并没有做好“有权”之后加以正确运用的准备。

组织活动的核心，在于权；权的核心，在于利。一部分人越来越怕谈权，是因为“利益”已经被越来越狭隘地理解为“钱”，因而谈谋利就变成了谈图钱。而权的责任跟钱一旦发生关系，就越来越说不出口了。

权之责，在于谋利。开公司总是要挣钱的，做一个挣钱的公司没有什么不好，但公司的利益却并不仅仅是钱。谈钱有谈钱的部门，例如销售部门如果也不谈钱，反倒不正常了。在整个公司的组织活动中，公司所追求的种种利益，都必将通过组织授权行为来明确。

这就是权、权责与权利的关系。

二

很多时候追问“凭什么”是一件相当无意义的事情。例如一个新员工可能会对部门领导充满置疑：他何德何能，凭什么来管我们？在这个新员工眼里，领导几乎在方方面面都差得很远：或比自己差，或比某些他觉得更尊崇的人差。这种追问显出理性的光辉，仿佛只要某人“有德有能”就应当居其位、行其权，似乎这才是公平合理而不受置疑的。

事实上他忽略了组织形成的代价，以及领导角色的成本。当把这些因素考虑进去之后，“有德有能”便居之，其实才是真正的不公平。举例来说，你可知那个“看起来平庸的人”事实上经历了多少轮组织变更与重建才留到现在？他在那些重大的利益选择面前，多少次坚定地与组织站在了一边？他又是在多少次外界诱惑面前，与组织不离不弃？事实上正是对“权、

利、责”的清醒认识，才使得他现在站在你的面前，成为看起来能力平平而又是一个“事实上的领导者”。

即便我们可以无视这些“看似是术”的东西，那么“机会与机遇”可能是唯一能解释大多数组织现象的理由。这主要是指，组织在早期形成时存有相当多的机会，例如职务空缺就更多；而在组织成型之后，这样的机会也就渐渐趋少了。然而参与组织早期建设的那些人，比如创业团队，他们承担了多少风险与责任，面临了多少困难与冲突，又岂是这些后来者可以想见的呢？

所以这个新员工要追问“凭什么”是既不公平也不合理的。但反过来说，组织内部如何解决权力与能力的问题，也是一个相当关键的话题。大多数情况下，“何德何能”也是组织内部任免时会提出的一个关键问题。但我需要说明的是，这是组织行为。组织行为是基于“组织的核心利益”而提出“何德何能”的问题。若无法看清这些利益，就无法判断组织行为；若无法左右这些利益，也就无法左右组织行为。这也是我说上面的“追问”无意义的原因：我们大多数的时候是在利益集团的外面，而非里面。

三

不过，你仍然可以有选择。

蔡学镛先生是我的前同事兼好友，他曾在博客中写道：

公司为我们做绩效考核的同时，我们也要为公司做绩效考核。要考核公司的哪些地方呢？薪资、股票、学习成长的机会、地位、成就感、稳定……这些都是评分点。你可以根据自己对这些条件的重视程度，做权重的调整，得到一个绩效值。比较各家公司的绩效值，做出去留决定。

如果你打算留在公司内，你也必须对你的主管做绩效考核。他是不是能充分发挥你的特长？他是不是答应给你的资源都有做到？他是不是有给你适当的奖励？他是不是赏罚分明？他是不是有在培养你？……如果你给这位主管打的绩效不好，你可以考虑换部门，或建议公司换主管。

所以你一旦选择了“在其位”，则应当抱有对“从整个公司到具体领导”、对整体组织系统的认可。这种情况下对于“凭什么”的追问，表面看起来是对别人的腹诽，实际上是对你自己选择的怀疑。你或是未能清醒地做出最初选择，或是难于坚定不移地支持这个原始选择，前者是谓“盲”，后者是谓“浮”。

首先要能责己、省身，这是修身的智慧。有了这种心态，我们再来谈谈这种选择背后的东西。回顾《大道至简》这本书，它在谈“画虎类狗”时就讲到过马援的家书。其实关于马援，我们还可以讲到他更为著名的一些故事。

西汉自刘邦始，历时两百多年之后被王莽夺了政权。而马援刚刚被王莽任为新成大尹（今陕西安康、汉中一带的太守）不久，王莽就被起义军杀了。一时天下大乱，四方兵起，于是马援就跑到凉州避难去了。这时地方军阀隗嚣占据了天水这个地方，却没有称王，而自封西州

大将军。当时，有公孙述在四川称帝，成都由之得名“白帝城”；又有刘秀在洛阳立了东汉朝，史称汉光武帝。于是在归附哪个朝庭的问题上，隗嚣犹豫不决，遂求计于马援。马援详查细思，认为公孙述功业未成却先已奢靡，不是王者；而刘秀恢廓大度有汉高祖刘邦的气象。因此他建议隗嚣归附刘秀，并随后随众前往洛阳，臣于刘秀。然而过了几年，隗嚣又有了反叛的意思，而马援则坚持自己对刘秀的看法，认为隗嚣想要称王是自挟奸心，遂帮助刘秀灭了隗嚣。此后，马援在刘秀一朝中西伐南征，战功无数。他在后世被追封为忠成侯，“马革裹尸”即是语出于马援。

从王莽、隗嚣、公孙述，再到刘秀，在得到刘秀起用之前，马援一直在选择，既有被动的，也有主动的。人的禀性不同，因此或建功，或立业，总是有着志向的不同。如同马援一生择主而事、守疆拓土，成就一代名将，这就是做事的人。

而在我们的现实中，往往一些人一时不得其志，或一时有所触动便谈“创业”。在遇到这样的人和这样的机会时，我们需要慎重选择。对于这样的人，小事尚且不能做好便谈“大业宏图”，不足与谋。如果不能与他“共其事”，又如何能与他“同其谋”呢？对于这样的机会，即便你身怀成事之大才，也应当先择业而事；即便你胸罗大志，欲成就一番伟业，也需知“要学会用人，先学会被用”这个简单的道理。所以我们要先谈事、后谈业，这才是行在实处；若是反过来，就是唱高调了。

反思马援的几次选择，有两处是值得回味与借鉴的。其一在于归附刘秀，是有眼光；其二在于平叛隗嚣，是会坚持。归结起来，明辨、坚持，便是从他身上可见的做事的素质。

四

马援志在建功，而不在立业。

但什么是“业”呢？现在一谈“创业”就是开公司，仿佛只要有个公司的名字，只要有个总经理的名头，就是创了业。归根结底，这还是在“什么是利”这个人生思考上随了大流的结果。“什么是利”是你对人生价值取向的发问，你追求什么，想要什么，然后想要通过什么方式来得到，这个思维过程决定了你的价值观与事业观。举例来说，你追求经济上的满足，想要更多的财富，并且认为得到它的方式就是开个挣钱的公司。这样一来，你创业的目标和方向就基本成形了。

我的确是在用一种粗浅的方式谈论“创业”、“财富”与“价值观”等问题。但这也是大多数人具有的一种思维方式，是普遍存在的，而并非少数的、精英化的思维、道德、价值与人生取向。我这样谈“业”的目的是要将这个话题赤裸裸地抛出来，并且强调：这没有什么不好，没有什么不正常，也没有什么不高尚。这是可行的、善的、正确的选择。

但这不是唯一的选择。“创业”其实是你的人生图谋，“一生所事者，为业”。因此，“求幸福”可以是你一生的事业，“图开心”也可以是你一生的事业，“谋财富”也可以，“得安心”也可以……历史上种种开“创业公司”的人物不少，例如张良开的“刺秦公司”，本以为杀了秦始皇天下就太平了；又例如刘邦开的“灭秦公司”，认为灭了秦朝，创建新的制度才可以给天下人带来幸福。当张良发现“刺秦”行不通时，就转而投入了“灭秦公司”。而我们在这里谈到的刘秀的“兴汉公司”，就是觉得天下要太平，就得四方安定，所以马援便有了施

展才能的空间。

在创业这个问题上，你的选择多多，例如张良就创过业，当然也失败过。但若是单单谈建功，要么出自机遇，要么就出自能力。前者我们暂且不谈。而后者，以能力论，需知有能力建功的人并不一定能创业，比如张良。建功者，既需要身具大能，又必须全力以赴。能与力，前者是内底里的素养，后者是外在上的施为。

这就是我们谈“事”的基础了。也就是说，你得有做事的才能，并且有做事的作为。马援两者都有，因而终成大器。以能而言，他首创了军事上的沙盘推演，是历史上最有名的伏波将军(1)；以力而言，他一生征战，最后病死军中。

五

马援之可叹处，在于他虽然成就了“马革裹尸还”的壮志，却也几乎落下了“死无葬身之所”的凄凉。前面说他“后世被追封为忠成侯”，讲的便是这个。

晚年，马援效廉颇请战，主动请兵去征讨武陵（属荆州，今长沙）的蛮夷。稍有失利时，刘秀便委派了素与马援不和的梁松去问责，并代为监管。但是梁松到部队的时候，恰好马援刚刚病死。梁松旧恨难消，便乘机诬陷马援。刘秀因此免了马援的侯爵爵位。马援的尸体运回，家人都不敢葬在原来的坟地，只草草埋了。然后家人用草索绑了自己，到朝廷请罪。刘秀拿出梁松的奏章给他们看，马援的家人这才知道蒙受了天大的冤枉。马援夫人知道事情原委后，先后六次向皇帝上书，申诉冤情，言辞凄切。光武帝这才命令安葬马援。

刘秀是一时失察？未必。但这件事就这样搁下了。过了一朝，到了汉明帝刘庄时期。这时刘庄评定开国功臣，但选出的“云台二十八将”中也没有马援。当大臣惊疑时，刘庄却只是笑而不答。难道又是刘庄一时失察？未必未必。

有趣的是，刘庄的皇后却正是马援的女儿。当年，刘秀一方面不再深究马援案的细节，另一方面又作主把马援的女儿聘为太子妃。后来，当刘庄为“云台二十八将”画像时，也正是他册封马援的女儿为皇后之时。每每两事并举，用意十分微妙。于是，这事情又再过了一朝。到了公元 75 年，刘庄死后，刘炟(2)即位。在他的支持下，大权在握的马皇后才终于为自己的父亲平反昭雪，追封为忠成侯。

汉朝的天下，究底里还是个“家天下”。所以事有亲疏，“家里”与“家外”的事是分别论的。梁松此人，其实是刘秀的女婿，因而梁松品行好坏，是否构陷马援可以暂不细论，此其一；他既是家里人，便是可以用来平衡日渐坐大的马援在军中的势力，此其二。马皇后在刘庄一朝中也不谈为父亲恢复名誉的事情，因为她深知这权力若失了平衡，就容易让刘庄心疑。再等到马皇后可得权柄，且军中再无马援旧部势力之时，这名誉恢复也就恢复了，再也无妨大体。

马援的得失往深底里谈，就成了一些人的权力政治。就好像我们再往深底里谈“凭什么”的问题，就成了办公室政治一样，这是我极力想回避的一件事。因此虽然我们谈到了“权”，但我却要就此打住，把权的问题放在一个组织的视角上去，而把接下来的问题，简单地归为“职”。于是可能就有人说了，权职权职，不都是一个东西吗？

其实不尽然。权谋的是利，职谋的是事，所以权有权益，职的利益却最多只谈到薪酬。

六

我们大多数人“求一职”，无非是“求一事”，求个做事的环境，有个做事的目标。但这个目标并不见得是公司的目标。开公司的目的是谋利，这是创业的基本思维：如果不谋利，你找一堆人来做什么？国家大事与私企小事，其区别只在于所谋的“利”的不同，谋“谁的利”也有差异；由此，进而有权，更进而有责。这些是组织形成的普遍逻辑。然而“职的利益”却非常明确，他只是一个人求生存的基本方法。

很多组织会将“组织利益”加于成员的“职责”，这是站在组织者视角上的愚民之术。我们谈到过，传统软件工程将根本任务与次要任务放反了，因为它将工程的根本任务设定为“抽象软件构成的复杂概念结构”，这终究只是学术的观点；而将它反过来，我们就会看到，“完成具体的一个工程目标”才是我们真实面对的问题。与此相同，组织者说大家的“职责”就是与公司一体，爱公司就像爱家，爱客户就像爱自己，这也不过是一种“教给你的”的追求；如果也将它反过来看，你就会明白，你的职责究底里不过是“做好手边的事”，以此作为自己讨生活的资本。

你做这件事的目的，首先是私人的，其次才是为公司的；而公司所希望的，正好是你做每件事都是为公司的，其次才是为自己的。我之所以一再提及这个话题，是因为公司的利益原本另有出处，公司与你谈利益，所站的都首先是公司的视角，而非你的视角。因此公司将你的职责，也预设成“为公司的利益”就是可以想见的事情了。

那么，你还为不为公司谋利呢？

职的业，在于谋事；职的责，在于成事；职的道德，在于忠其事。这才是职业的视角。你的“利”在于你决定“做这件事”时的选择（例如薪酬，又例如人生理想），这是首要目标；若因这件事而成就了公司的利益，却是次要目标，或者我常常称之为附加收益。

但我仍然必须强调，这一问题在创业与守业阶段是略有不同的。你如果加入一个创业团队，你加入时所设定、所承诺的这件事的目标便是创业，是将自己的利益与“所创之业的利益”放在一起的。若非如此，你便仍然称不上创业，只不过是打着创业的名头，求着自己的私利罢了。创业与守业之不同，就如同攻城与守城的不同。你看攻城，所有的人都朝着一个目标，撞着城门，爬着城墙垛子，前仆后继总为着相同的目标：进城去，就什么都有了。而你再看守城，各自守着各自的城门，管垛口的只管垛口，管马棚的只管马棚，每个人都看着自己的职责；若管马棚的不管马棚，而跑去守垛口，马群散了在城里，撒开花儿地跑，既乱了军心也挡了他人的事。于是管马棚的就被砍了头，失了职也失了命。

守城之重，在于各司其职，有条不紊。尽管头儿会告诉你说，丢了城我们大家都玩完，但是你也知道：人家打进来不一定会杀了你，但如果你现在失了职去守垛口，那就可能立即被正了军法。所以从守城之职的视角来看问题，无论于私于公，其最优解是看着垛口一个个失掉，又或者被夺回来，而自己仍只管好自己的马棚。所以管好马棚是你的首要目标，如果这促成了守城的胜利，那是附加的收益。

再说“忠其事”。其实，在我看来，这已然是非常高的道德标准了。

西周时期，齐国的崔杼杀死了国君，于是齐国的史官就写道：“崔杼弑其君。”这是说崔杼不忠，以下犯上地杀了他的君主。于是崔杼不满，就把这个史官给杀了。钱穆先生曾在这里点评说(3)：那时周王室派出很多史官，他们虽在各（诸侯）国，而其身份则仍属王室，不属诸侯。因此，崔杼可以把这个史官杀了，但不能以齐国的名义另派一人来做。于是史官的两个弟弟便接任兄长的职务，再来照写“崔杼弑其君”，崔杼再把他俩杀了。又再换第三个弟弟，还是写成这样，崔杼没有办法了，只好住手，不杀了。

当时的史官受命于周天子，在齐国称为“大史”，而在齐国以南的其他诸侯国，则泛称为“南史”。所以当时有“南史氏”听说齐国的大史都因此死光了，于是便自备了录史用的竹简，捧笔前往，以志赴死。

因为受命于“录史以存真”的职务，便舍了性命也不改一字，便舍了性命也要前往记录史实，这便是我所说的“忠其事”。

七

你需要一个做事的团队，但事实上这非常难得。

我们这里讨论的团队，并不是公司（或我们统一地称为“组织”）中的一个部门，至少并不简单地是这样一个组织结点。组织中部门的规划取决于许多因素，有些时候与企业的历史背景有关，例如国企或事业单位下属的一个软件公司；有些时候与企业领导者的管理风格有关，例如某些采用家长式管理的私营企业；有些时候又与企业正在经历的变革有关，例如业务转型阶段中的企业。在这些种种不同的背景或时间点中，一个“具体做事的”团队有可能保证它的相对独立吗？

答案是：几乎不可能。也就是说，在不同的组织结构、风格与背景下，我们事实上难于得到具有相同或相似性质的“做事的团队”。但是出于两个目的，我们不得不假定：

所谓“团队”，是具有相对独立的组织特点，有着自我目标和道德环境的一个特殊群体。

这两个目的是：其一，我们在讨论中需要一个相对纯粹的模型；其二，构建新的团队，可以作为“立一个项”的普遍条件，即我们有望针对项目来构建团队，而在一定程度上忽略既有组织的影响。

事实上现实中的大多数公司都已经踏出了这关键的一步。大家似乎都已经意识到，没有哪家公司一开始就是围绕“工程化的软件开发”来构建组织的，也没有哪家公司是“先建设好组织模型”再来生产与经营的。在现实中，大家都是边干、边扩张、边调整，组织是在动态发展的过程中渐渐感到“工程化”的必要性。但这种必要性又与现有的组织形态存有一定的冲突，例如：

□ 其一，敏捷工程通常要求团队的规模较小，但现有组织中的任何一个部门都可能有上百

人；

□　其二，如果以“小组”为单位来应对敏捷工程，就会发现小组中存在许多的角色缺失，例如测试角色可能无法由直接的团队成员来担任，因为这些专职人员可能归属于公司内的另一个独立部门；

□　其三，当一个工程团队所需的资源零散地分布在公司的既有组织中时，“团队是什么”就成了一个非常实际的问题，而更实际的问题可能还包括“团队到底有哪些责权”。

所以上述的假定是有必要的，这有助于通过组织授权来建立“具有明确责任的团队”。基于这样的假定，可以大略地将“围绕项目目标而组织起来的一群人”归纳为三种模型：产品线团队，对应于企业业务；产品团队，对应于客户产品(4)；创新团队，对应于自发需求。

三种模型的根本相同点在于：一群人，一个目标，一种方法。(5)

第二节　你要什么

一

我一直有一个问题：既然是“前人栽树、后人乘凉”，那么前人该不该栽树呢？

栽树事实上是一个布局的事情。有两种方法来完成这个布局：第一种是直接移植一批树来；第二种则是新栽一批树。这两个布局的结果可以是完全相同的：一片可以乘凉的树。但是，两者在本质与操作手法上都存着极大的差距。

移植的方法是管理层在组织问题上最愿意采用的，但考察收益，其风险却又是最大的。简单地说，有冒险精神的管理层往往采用这一策略。所谓“冒险”，意味着这样一些事实：

（1）管理层对移植目标并不充分了解；

（2）移植目标的可能收益相当大，从而能构成诱惑；

（3）可能发生的移植失败在一定程度上来说是可以承受或有所预期的。

例如，我们可能要在开发活动中引入“敏捷方法”，于是我们要考虑建设“敏捷团队”。但管理者可能并不清楚什么是“敏捷团队”，只是听说这样做开发快、质量好，而且对技术人员的提升很明显，于是便有意让现有组织“引入”这一组织方式。

如果敢于冒险而又不敢于坚持，这种移植方式往往是收效很差的。但正是因为管理层原本就

是期望“快速地”解决问题，所以他们在这件事上并没有足够的毅力。一旦“移植组织”面临团队内部矛盾，或团队与现有组织结构产生冲突，整件事情就会渐渐陷入泥沼。接下来，管理层如果在“继续与否”这个问题上稍一犹豫，则整件事情就要被搁置了。这种“稍一犹豫”的根本原因在于，组织管理者原本并没有做好“管理这样一个移植过来的组织模型”的准备。也就是说，他们事实上根本不知道怎么去管理。但这样的管理层与“山大王”还是有着不同，这一区别在于：前者缺乏对组织的认识，后者缺乏对权力的意识。

敏捷团队实践中最大的问题，并不在于其内部运作——这通常在短期的训练之后即可解决，而在于这个团队与既有组织模式的整合。更明确地说，就是让“旧的管理层”如何接纳新的事物的问题，而这并不是决定“移植”就可以解决得了的。因此，在诸多的尝试之后，（保守而又不思上进的）管理层往往认为他们已经“迫不得已”要选择第二种方法了，即栽一片树，这总好过弄一堆树来却不知道怎样让它们成活。

栽树是相对慢得多的一种布局，因此“为什么栽树”就成了很实际的问题。急着乘凉摘果的组织是没时间栽树的，这在逻辑上也成立：人都快饿死了，还谈栽树做什么？所以一个创业型的组织通常对这个问题缺乏思考，而当整个组织的规模达到成百上千人时，这个问题就凸显了出来。而且它一露出端倪，就将立即伴随着大量的组织变更，例如部门调整、人事任命、责权划分、奖惩措施、问责制度等。

这些都只栽树的手法而已。树可以拔了再栽，只要地盘还是你的，就无碍大局。这就布局的根本思路。时、势造英雄，栽树就是一个让中层成长起来的、长期的组织战略。

二

“木秀于林，风必摧之”，只要不倒，长成歪脖树也是景观。在这个过程中，大多数人只着眼于歪脖树“成长的奇迹”，或感叹左一阵右一阵的风，少有人观察到那片“林”。

我在 D 公司的时候，例行有三个月一次的组织管理大会。某次，CEO 宣布了一个大胆的组织计划：

（1）建立产品专员制度，产品专员可自荐或推荐，由 CEO 亲自挑选与培训，适当的时候由总裁办公室委任到各生产部门；

（2）扩大管理规模，在原有三大中心的基础上，增加两个中心和相应的部门组织结构；

（3）宣布成才计划，包括购置与投资第三方开发组、研究室或小型软件公司，并成立相应的监管部门与监理机制。

大会结束时，许多人在私底下表示了对这件事情的不理解：我们才两千人，怎么公司就有了这么多管理层级？干事的全变成了当官的，这活还怎么干？当一个同事向我提出这个问题的时候，我回答道：如果你把这看成一万人的公司的管理规模，你就觉得合适了。

CEO 是在为一万人的公司做组织布局。

三

为什么别人做这样的事情就是乱来，而 CEO 做，看起来就是“组织布局”了呢？先撇开“该谁做”的责权问题不说，所谓布局是“有象无形”的事情，即我们大体上看得到一种趋势，并在这种趋势下采取了一些作为，但这时确指“做成了什么样”还言之过早。

这次组织布局概括来说，就是三点：

（1）权力下放的核心方法；

（2）组织扩大的规模与路线；

（3）新组织的内容来源。

在组织上，“人与事”是两个核心路线，而“权利”则是决策依据。正如我们前面所讨论过的：“权利”围绕着“利”的确认而构建，“目标”是对“利”的阶段性设定，“人与事”是实现目标的组织要素。

在 CEO 的这个组织布局中，你看得到“基于产品的专员制”，其核心在于“权力”由 CEO 给出。因此产品的定义权、决策权和认可权等，都归“CEO + 总裁办公室”的组织结点。换言之，对于今后上万人规模的公司来说，其核心利益以及与此相关的产品生产决策，将仍然是由 CEO 来把握。

你也看得到“新中心与新部门”，这意味着“项目中心管理制”将是后续组织制度的基本模型。你还看得到“购置与投资”，这意味着新的生产力量主要来自于外部，并且公司为这些新的组织结点配置了“监管部门与监理机制”。这些都必然意味着“大量的、非生产性质的管理岗位”的产生。

那么现在来看，这个“组织布局”还有冗余的问题吗？

再深一层地发问：你看得到其中的组织机会吗？

四

但这与“具体工程”有什么关系？

工程，或用来指代“其具体事务”的项目，是中层管理语境下的一个讨论对象。对系统，它依赖于具体的组织结构形态；对外，它依赖于管理层授权；对内，它依赖于团队协作并逐渐形成独立的行事风格；对局部，它依赖于产品或产品线对于“事务”的定义、限制以及相关的决策过程。

具体工程，是一个无法脱离环境约束的、对工程的定义。大多数对具体工程的成败构成显著影响的因素，都是组织行为而非具体事务的做法。

因此，如果组织在布局，那么项目多数是“求势”而不是“求实”。我曾经举过一个例子，你的主管可能会要求你(6)：

□ 在 Lucene 这个搜索引擎的基础上做几个示例程序。

但他不会告诉你：

□ 公司打算选择一个开源的搜索引擎。

或者更进一步的问题是：

□ 公司必须在使用商业产品、开源代码和自主研发一个搜索引擎之间做出选择。

再或者导致这一问题的根本原因是：

□ 某某搜索引擎公司正在与高层探讨战略合作，而我们必须在“尝试合作”的同时，通过支持某一个开源引擎来保持战略选择上的灵活性。

如果主管所要求的“做几个示例程序”是一个项目，那么项目中的实施者又如何能看到组织在“布局”上的全像呢？这个例子的不恰当处在于：也许有人会认为“做几个示例程序”是可以放弃的棋子，而布局是那个弃无可弃的棋眼。

但除了布局者，谁又知道棋子与棋眼的区别呢？

五

组织的契机，便在于棋子与棋眼的选择与变换之间。

在 E 公司，员工 A 在跨部门调动时，新部门的主管 B 找他谈话，大体上是讲讲今后的工作内容、工作方法以及相互之间的协调配合等。但他们的谈话，事实上已经快速地切入到今后工作的责权分配问题。当时员工 A 便提出了他的要求：你可以给我一个目标，我来决定怎么做并确保它的推进。但是有两点：第一，不要怕我闯祸；第二，我做事可以，不背黑锅。

主管 B 对这个“赤裸裸的要求”感到有些震惊。然后他从职业精神与工作态度的层面对此加以了高度赞扬，并接受了员工 A 的调动。在随后一年多的时间里，员工 A 既没有闯祸，也没有背黑锅。

但问题是，员工 A 基本也没什么事可做。

不做事当然不闯祸，当然也不会扔出黑锅或背上黑锅。主管 B 的用人与治事是否出了什么问题，这并不好说，毕竟这与“部门要做什么事”以及“员工能力范围内适合做什么事”还有关系。但是对于员工 A 提出的“条件”，主管当时的反馈并不适当。

当我们展开一个合作关系（而非简单的雇佣关系）时，总是有一个相互试探过程的。员工 A

的条件看似随意或只是为了彰显个性，但实底里是对管理责权的要求，如果他与主管 B 之间的确可以形成权力关系的话。因此大体上来看，这个要求的目的是在合作、任用、使用等多种相互关系上定一个基调，例如期望借此了解“如果要让我当排头兵去做事，那么限制的边界在哪里”。

对这一问题，我认为可期望的、也可能是最佳的答复是：你要做任何事，我都表示支持并完全不发言；但是如果我要发言，那么你一定要听。

棋子与棋眼的关系，便在于可控与不可控。如果主管不能控制一枚棋子，那便将他只用作了棋子，而绝不用作棋眼；若要将他用作棋眼，那便要放得下去、留得住、看得到。

主管 B 对员工 A 的反馈只着眼于“可否成事”，而不在于是否“以之成势”，前者是用作棋子，而后者可作棋眼。但在他们所讨论的组织层面，仅仅谈“事”是不够的，信任、责权、方向以及对形势的分析与把握等要达成一致，或形成一体，才能谈得上合作，否则便难成大用了。

六

一个九岁的小学生说她“幸亏没有要当总统的梦想”，于是妈妈问她为什么。她说：

当总统也就是刚刚当上那会儿会很开心，好像是赢了。你看奥巴马，选举的时候许多人支持他，那么风光，而一当上总统，反对他的人马上就出来了。(7)

没有人天生就是管理者。不管是老板、部门负责人或项目经理，大家只存在管理对象的不同，在管理能力上其实是一样地“先天性缺乏”。所以多数情况下，我们都处在一个“并不太合适”的组织位置上，至少我的所见是如此。因此我们必须清楚地认识到什么才是“合适的位置”，或与这个位置匹配的“合适的能力”。然而在这一点上，绝不能以“别人的反对与支持”作为判断标准。

对于管理层来说，一个管理者是否“被反对或被支持”都不是组织任免的核心原因。大多数有成就的管理者，往往都面临爱恨交织、毁誉参半的局面。通常一个组织内的各个角色，以各自的视角与利益取向来看这个“管理者”，损己则责之，利己则爱之，反而很少去思考“如果自己也在那个角色上，又能如何”的问题。

很少有人会设身处地地去“为别人思考”，所以大多数的言论貌似“诚实、诚恳、由衷”之言，实则是无关痛痒的“观感”；即使这种观感有一点点进步，也不过是因为触到了发言者自己的利益而变得带有些切肤之痛的味道来。

九岁的小学生还不知道那些被代表的利益，因此只看到了支持与反对之间的矛盾。

七

工程做不做得了，是不是技术问题？

这是一个关键的设问，涉及很多方面。例如其一，工程在时间与空间上可能达到的规模，是否可以通过组织团队或项目来满足；其二，工程中可能出现的问题，是否可以在技术上解决；其三，工程所需要的资源，是否可以持续供给。第一个问题，是工程在系统架构领域中的技术问题；第二个问题，是工程在产品/业务领域的技术问题；第三个问题，则是工程背景下的商业环境问题。

工程自身的技术问题，是类似于“过程、方法、工具”这样的、传统而学术的工程视角下的问题，是把工程作为技术并对其中所涉及的干系者（人和物）的统一管理。但是这种管理又具体与上述三个方面（架构、技术、环境）的问题无关。例如，我们在工程中所讨论的过程是以瀑布模型（分析－设计－开发－测试等环节）为基础的，它根本上与“怎样一个具体的架构、技术与环境”无关。这也是《人月神话》一书中“银弹”的核心，即我们是否可以找到一个与具体情况无关的工程方法。

这在根本上，就是“唯技术论”的思维。而一个具体工程下所需的管理与组织，却并不能基于“技术思维”而得到。正如我一再强调的：

真正的管理者，不看“管理的技术书”。

管理的总的决窍，在于一以贯之。无论是什么，吐啊吐啊就习惯了。(8)就如一个著名的组织行为观点所说：改变不了思想，就先改变行为；行为改变了，慢慢地思想也就改变了。对于大公司的体制来说，往往需要“我们”是被改变的那批人，并且往往是“先从行为改起”。在本质上，这是形成企业发展的一种势态，当大家的行为模式、方法、习惯以及思想趋同时，要做的事情便可以顺势而为了。

真正的组织管理者，既在寻求组织的不变性，又在这种不变性中看到变化的机会。组织中的大多数人只是被左右挪动的一枚职务角色，甚至像刺秦的力士一样连名字都不为组织所知。正因为如此，探索并形成有利于“工程实务”的组织结点，是大多数公司在“软件研发管理”上的痛中之痛。

在这样“探索并形成的某一个组织结点”下的一个工程（或项目），便是我所谓的“具体工程”(9)。这个工程具体而微：有人、有事、有目标，也有可资借鉴的过程、方法与工具。这个工程借鉴但并不照搬任何一个既已存在的模型或过程方法论。这个工程的首要任务是完成具体的“某件事情”，这既包括让大家清楚地认识到这个事情的目标（可能表达为产品或产品线上的阶段等）、收益以及风险，也包括构建让所有成员都能有效投入的职业氛围。不过，它并不为“明天公司是否以此为范本”而负责。

那是别人要的，不是你要的。

————————————————————————

(1) “伏波将军”是一个名号，意思是“降伏波涛”，因此历史上有很多将领被任命或被称为“伏波将军”。

(2) 音“达”。

(3) 引自《中国史学名著》，钱穆著。

(4) 这里的“客户产品”与前一种的“业务”是不同的。这里用“产品”来表明“有明确的特性内容的需求”。所谓“客户”，既可以是以企业自身为目标，也可以是以特定用户或用户群为目标，或类似外包中的出包方等。

(5) 这让我想起《集装箱改变世界》中提到的著名的“3C 原则”：一只集装箱（container），一个客户（customer），一种商品（commodity）。

(6) 引自《程序员修炼之道——从小工到专家》，第 7 章。

(7) 引自《羊城晚报》（2011.07.28）的文章，《日本孩子长大后最想做什么》，作者唐辛子。

(8) 这既是温水青蛙式的风险，却又是组织行为的精义。

(9) 我们此前的许多讨论并不适用于“开源工程”，但这句话是绝对的例外。具体工程是一种从对“工程的结构元素”的认识出发，试图在特定背景下形成适宜的工程方法的思维方式，本书的讨论可作为这一思维的一个示例。

附录一

行在道上，从局部到全局(1)

EHM 不是一个用于实作的工程模型，它只是从某个角度分清了工程的一些环节而已。如同“牛屎图”一样，EHM 是一个思考模型。相较而言，EHM 更易于发现问题，而“牛屎图”更适于理清思路。如果非要从中找出个“更核心”或“更重要”的，那么在“牛屎图”中要以哪一个作为最底层的圈子也成了问题。所以不妨抛开“谁更核心”的问题，把赵善中先生(2)的模型、EHM 图以及“牛屎图”作为可以相互补益的图来看，这样就所得多多了。

一个软件产品，究竟是被“开发”出来的，还是被“架构”出来的，抑或是被“管理”出来的？这是一个争执不休的问题。一般人，尤其是技术出身者，会直接否定掉第三种答案。他们认为“一个软件产品不可能被管理出来”，它只能是被开发出来，而管理不过是这个过程中的官僚角色罢了。但什么是管理？管理真的就是今天命令你工作、明天要你汇报进度的那个人吗？是不是我们把一些个人的私怨——对某个拙劣的管理人员的不满或轻视——带到了我们讨论问题的语境之中？

开发、管理和架构三种角色，站在自身角度，都会认为自己应该主导软件产品产出。这种观点我向来持以理解而又置疑的态度。更渐渐地，我对所有类似“某个单一角色主导了软件产品产出”这样的观点都表示怀疑。正如高焕堂先生(3)所提到的，这更像是一个“三足鼎立”的局面。这种“鼎立”的局面是一个衡势，这意味着它的平衡是瞬态的，且总在平衡与不平衡之间。所以我的疑问是：这种局面对于软件的研发、项目的过程来说，是适宜的吗？应成为常态吗？

从组织学（而非单纯含义的管理学）的角度上来说，鼎立是组织的一种形式，是管理的一个施为目标，而不是管理本身。虽然我认为高老师所指出的“鼎立”的局面可能是将来的方向，但如何去组织这样一个组织，管理它，并在这一局面上产出软件产品，是更进一步的学问。

在《大道至简》一书中，我基本否定了对软件开发过程的管理：

“开发团队并不需要管理。或者说，在你没有弄清楚状态之前，不要去管它。”

同样地，我也否定了传统的软件产品的“产出”观念：“工程不是做的，是组织的。”既然我们不能“管理”一个团队，又不能去“做”工程，那么我们该怎么办呢？在我跳出上述的三个角色之后，我得到的答案是：组织。

进一步地说，管理角色的任命、项目团队的结构等，都是在“形成组织”——这一过程中的产出和阶段性的决策。对于（泛义的）软件项目来说，没有一成不变的组织，也没有一成不变的模式。更进一步地说，在（具体的）某个软件项目中，组织的行为也处在变化之中。

我推崇高老师“以序容易”的架构思想，而这也意味着我上面的言论会有一个推论：必然以及必须要存在一个“无序”到“有序”的过程，即我们最终仍然会得到一些“组织模型”（以及管理、过程等模型），用以规划和指导实施同类性质的项目。我不否认这一状态和阶段性结果的必然存在，但我怀疑现在是否已经存在，例如某些工程界吹嘘的“模型”是否就是终极的银弹。

因此，回到赵老师的话题，在“工程实施”的语境下，究竟（现在）有没有确定的模型呢？我不置可否，唯只置疑。架构是工程中的推手吗？是推动的原力吗？我从架构这一角色的位置及权威性上看得到希望，也从这个角色所必备的能力涵养上看得到希望，但是推及到组织及具体的项目，架构角色真的有这样的能量吗？谁赋予了他调适组织形态、降低组织内耗的责任？如果他没有这样的责任，那么组织如何生存？如果组织不存在了，岂不是仍然回到了“皮之不存，毛将焉附”的问题上了？

————————————————————————

(1) 节选自《程序员》杂志 2009 年第 10 期同名文章。

(2) 赵善中先生是“结构行为合一”架构方法的创始人，人称“SBC 架构之父”。阿拉巴马大

学计算机与信息科学博士，先后任职于中华电信公司、GE 实验室、中山大学资管系。时任国际企业架构师协会台湾分会理事长。

(3) 高焕堂先生是台湾软件架构设计大师，从事 IT 行业近 30 年，被称为“台湾 OO 技术教父级代表人物”。曾创办 MISOO 对象教室及《SoSE 杂志》，时任 MISOO 软件开发与管理顾问公司 CSA 职务。

附录二

本来面目——大教堂、集市，与作坊(1)

不管是理想化的，还是神化的方法与论断，在《梦断代码》里都用到过。当然这些方法、论断也确如乩言一样灵验或失灵过。正因其灵验，以及偶尔的失灵，才让那些崇信者拜服与生畏（更何况一些崇信者原本就是这些方法与论断的始作俑者）。

《梦断代码》中的团队有一个绝对集市的名字——OSAF，也有绝对集市的原则：开源且接受大众的眼球（眼睛足够多，缺陷无处躲）。但是这个项目启动了一年之后，这个团队还没有一个真正意义上的领导者：他们（包括那些从来不出现在团队中的志愿者们）总是在不停地开会、讨论，以及推翻上一次的开会与讨论。

团队和产品的精神人物，是又出钱又做事的卡普尔。他声称要做一个能打败一切 PIM 软件的 PIM 软件。简单地说，直到这个软件只剩下以自己为对手，这就是目标。类似的目标也出现过、成功过，例如 Internet Explorer。因此，看起来卡普尔是要在开源界打造一个类似的神器。正是在这样的精神感召下，一批热血之士“混成”（混然天成的混成）了这个团队。

卡普尔看着大家积极地讨论着他的“原始需求”，并接受了他们制造出来的更多的需求。在这些目标堆积成山之后，卡普尔开心地聆听了第一任项目经理基尔默（或只是一个技术首领）的预期：我们将在年底发布该软件的一个早期版本，预计明年年底或稍迟的时间即可以发布 1.0。

“他的预估太过乐观了”，Rosenberg 在后来说。

在《大道至简》一书中，我写过：做了这么多年项目，当我一听到“那我们就开始开发吧”这句话，就哆嗦。因为大多数人在说这句话的时候，连组织结构是什么都还没搞清楚呢。

果然，接下来 OSAF 团队陷入了“与组织作斗争”的阶段。没有决策者，这意味着任何一个“看起来像决策的东西”都可能在你背转身去的时候被打破（P.154）；没有设计者，这意味着任何“看起来应该已有雏形的东西”其实都未曾现身（P.132）；没有跟踪与限制，这意味

着任何人都可以停下手中的工作，去做另一件“看起来更要紧的工作”（所以他们制造了大把大把的工具并不停地废弃它们，P.129）；没有……是的，OSAF 团队没有任何可以依赖的组织结构，直到一年之后，他们把项目组分成了三个小组，并设立了正式的团队经理，由卡普尔兼任（P.157、P.158）。

“别再继续发疯了！在书里，他们该错的都错了！”Rosenberg 在后来说，“我自己也在怀疑，他们何时才能起步？要花多长时间？障碍是什么？”（P.158）在这个时候，《梦断代码》这本书正好写到一半。你可以猜测另一半的结果，或者干脆直接给它判死刑。

但真正的问题是在哪里呢？是集市工程的失败么？问题是，《梦断代码》在后面还记叙了微软 Longhorn 项目的失败，那里有着最具规模的教堂队伍，从神职人员到扫地的，一个不缺（据说连扫地的都至少是硕士以上学历，以备将来发展为扫地老僧）。

具体工程也会失败，给一个工程冠以“具体”的名字，并不代表它就成功了。《梦断代码》讲述了一个具体工程的失败，失败的过程闪耀着光辉：集市的、教堂的，以及牛人们的、有钱人们的……无限光辉。

但是他们想做什么来着？

————————————————————————

(1) 节选自《程序员》杂志 2009 年第 3 期同名文章。

附录三

杀不死的人狼——我读《人月神话》(1)

总的来说，我事实上并不反对某种具体的方法，而是在努力学习种种方法。但我并不认为有什么单一的方法能解决所有问题。每一种解决方案都有其前提和背景，精通一种或全部工程工具、方法和过程，都无助于掌握这些前提、背景以及潜在的关系。理解工程的适用性，涉及工程专家的整体素养和实践经验，也涉及具体的工程环境、目标和实施者（团队）。而这些课题，正好是目前的软件工程甚少谈论的。

在银弹的话题上，答案不外是：有、没有和将来一定会有。我的答案与这些都不一样。我认为“有没有”的话题没有意义，不值得讨论。因为 Brooks 在人狼的设定上是一个伪命题，而在银弹的假设上则是学术的。我进一步的观点是，广义工程对首要任务（人狼）和完美方

案（银弹）的假设，不应该成为狭义工程所追求的终极与利器。

我们要分清狭义工程与广义工程。正是由于他们对本质需求的设定完全不同，因而也有各自不同的主要与次要任务。一切工程活动的历史告诉我们，曾经的经验、失败和成功，以及据此在广义工程中归纳推演的理论，都只是我们在具体的、狭义的工程中的参考，而非无往不利的银弹。

曾经，我们如同走在一片沙漠，看不到沙漠的边缘，也不知道如何行走。Brooks 的伟大贡献，在于他用《人月神话》指出了一条道路并教给我们基本的行走方法，让我们从中学会了辨识流沙，懂得了沙暴的征兆；同时“没有银弹”的假设激发了我们的斗志，如同有人在说“你要证明沙漠有边界，你就拿出一束青草来看看”。

于是有了几种喋喋不休于“有没有青草”的人。一种是已经在灵魂上上升到另一个层次，故而无需行走于沙漠。另一种则只是站在沙盘边上，用长杆推动卒子，而鞋子上根本不会沾上一点点沙土。

还有一种人则不时地宣称他们已经找到了青草。他们总能确保你在付费的时候看得到青草，而你转身时才会发现那里并不是沙漠的边缘。

剩下的人一边喃喃着“沙漠边缘的青草”，一边在焦燥的沙海里缓步前行。不同的是喃喃而不自觉者掉进了沙坑，心怀憧憬而盯紧脚步的人则走出了沙漠。

走出来，你才会觉得：原来有没有青草，并不是那么重要。

————————————————————————

(1) 节选自同名博客文章（2007 年 3 月），该文章是关于《人月神话》的读后感，是“具体工程”的滥觞。

引言

简单的本源

接下来，我们将要提出一些现实中的简单问题。这些问题是如此的简单，例如为什么要用一对大括号“{}”来将代码括起来？之所以说它简单，是因为你会看到每本书都这样忠实地写着，而且你的每一个老师、每一个技术同行，以及每一个代码范本都如此教导着你。而它又

无比复杂，以至于到了一种名为 Python 的语言出现时，我们就再也想不明白：为什么代码不再包含在一对括号之中了——对于许多人来说，将这样的一对括号换成一组“begin .. end”都是莫大的挑战。

编程的世界是如此的奇妙，如同我曾说过的一样：会不会编程，甚至成了某些人的智力评测基准。然而也如同上面所谈，解释其中某些看起来习以为常的现象，既大有必须，亦必为挑战。

那么我们有没有办法，不使用那些艰涩的公式或者数理逻辑来证明这个世界的必然性，而仅仅只是去说明它呢？

我想，代码总是需要拿来被“阅读”的吧。先不管读者是谁，一个东西要被阅读，它至少要具有两个性质：能被叙述，以及能被记载。这其实涉及了两个更深层次的问题，前者是要求有一个语言系统，后者则要求有一个存储系统(1)。语言系统的特性取决于对话的双方，而存储系统的特性则取决于存储的对象，以及存储的条件……

等等类似这些的、将在本编中出现的文字，在我看来就是我们对“代码为什么是这个样子”的本源性的思考。那些完全不了解，也不探知，甚至未能察觉“本源问题”的人，努力地清扫着地毯上的灰尘并一遍又一遍地检视着，点着头发出赞许的啧啧声，而无视于空气中漂浮的粉尘——即使它们在半个小时后又将掉落在地毯上。

所以一个不能思考事物本源的人，是不可能具有开创性的，他既不能解决问题本身，也不能发现解决问题的可能途径。

————————————————————————

(1) 这其实是人类文明的根本，通常含义的文明是从有文字记录开始的。

第四章

计算系统

计算机本质上来说仍然是一种算具，其基本构造与历来的算具并没有什么不同。其核心仍在于对数和算的表达，即一种表示与存储数的方式，以及一种计算的方法。

计算本身对工具是不存有依赖的，例如心算。但计算的复杂度，使得我们将计算过程进行切

分成为必须，这基本上可以视作函数这一抽象概念的由来。

第一节　数，以及对数据的性质的思考

一

小的时候，我的数学成绩不错，但每次考完试回家，父亲总是问我：“算数考得怎么样啊？”因此有一个问题让我很是苦恼：为什么我的课本是《数学》，而父亲问我的却是“算数”呢？

很多年之后，当我有足够的资料来追溯这个问题时，发现在我父亲的年代，他们所学的那门功课的确是叫《算数》，后来有些书变成了《算术》，另一些又变成了《数学》。尽管我理解了父亲用这个词所基于的教育背景，但我仍然感兴趣于上面这几个词的演变过程。

再后来，我终于了解到连我学的历史课本也出错了：我国已经证实的最早的数学著作并不是汉代的《九章算术》，而是早了两三个世纪的《算数书》。这两本书的名字都是可查证的，前者为历代记载，后者则写在出土的竹简上。于是我终于了解到一个事实：古人其实最早是将这门学问理解为“算数”的，再后来才退步了，理解为“算术”。

为什么说理解成“算术”就成了退步呢？因为将书名写成“算数”的人，还知道我们“算”的对象是数，而作“算术”者，便只当这是一门为算而算的“术”了。

很多很多年之后，我们开始学计算机。很多人花了许多年功夫，最后尽在“计算”上做足了花样，却忘了我们原本算的仍然不过是“数”。算这些数的那些算法，只是“术”而已。

“算”是程序之表，“数”是程序之本。

二

计算机系统的数与算，是基于数学与电子学而发展起来的。

首先，这里的数（number）并不复杂，我们也都知道是所谓的“0”与“1”。在这个“数”上的算法也很简单，是所谓的布尔运算。它们作为二值数的提出与基本运算的确立，基本可以视为是布尔在 1847～1848 年间的主要学术贡献。90 年之后（1938 年），香农在提交给麻省理工学院的硕士论文中，展示了布尔逻辑在电子学中的应用。又过了 10 年，香农首次提出使用 bit 这个词来表示二进制数字。(1)

bit 这个词被创造出来，用于表示这样一个二进制数所能代表的信息量。“由于 1bit 表示的是可能存在的最小信息量，那么复杂一些的信息就可以用多位二进制数来表达”。(2)

信息，是有意义的数据(3)。

三

数据（data），并不是数，而是数的系列。数学概念上的布尔数，是纯粹抽象的，可以是黑白点，也可以是正反面；电子学概念中的布尔数，是逻辑门电路中的高低电平信号。而数据，其内聚的特性表明它由一系列存有相互关系的数构成，其外延的特性表明它可以与其他数的系列建立新的抽象关系。

这里的“系列”，只是对数据的构成元素（例如多个二进制位）之间存在着的关系的一个简单称谓。这种关系是什么呢？例如，我们说“A”字符是一个数据，即是说它是由“1000001”这七个二进制数形成的组合关系。这个组合关系是强约束的，如果换一个组合，则不是“A”字符而是别的字符了。再例如，我们既然说“A”是“1000001”的组合关系，则也同时是说“B”必然是“A”的组合关系的一个外延。但这样的外延存在多种可能性，对于这两个字符以及由此构成的数的系列这一概念来说，其数学描述方式为：

我们知道，在“ASCII 字符集”这个系统中，f()所表示的外延关系是 inc()(4)，即 B=A+1。而在另外的某个系统（例如密码卡）中，f()则可能是另外的一个函数。数，与数据之间存有的不确定的函数映射关系，是我在前文中用“数的系列”，而非“数的序列”的真实原因。

我们所面临的硬件系统以及其背后的逻辑基础，都是基于数的概念；而我们的软件系统，例如我们使用某种开发工具、语言，或者操作某个设备的指令集，都是基于数据的概念。因此在软件开发（而非计算机研制）中，第一个要明确的问题是：

我们要操作的数据是什么？

四

总的来说，我们的计算机由五部分构成，即控制器、运算器、存储器、输入设备和输出设备。然而这五个部分所理解的数据并不一样，以个人电脑系统为例，见表 2-1。

表 2-1　PC 中的设备与其理解的数据

我们还可以举出更多的计算机组成部件以及它们对数据的不同理解，例如显卡、打印机、硬盘，以及扫描仪等。但归总起来，这些“数据”只有两类：

□　一是在设备与设备之间，我们确立了以传输总线带宽为标准的数据交互型式（type）。即，设备与设备之间的数据，明确以如下单位传输：位（bit）、字节（byte）、字（word）、双字（dword）等。

□　二是在控制与被控制对象之间，我们明确了以交互行为（action）为视角的三类数据(5)，即逻辑（logic）、指令（instruction/operator）和操作数（operand）。

正因为交互行为本身亦是（可以按照确定交互型式来定义的）数据的一种，可以在设备与设备之间传输，所以我们的击键行为最终可以表达为某些 CPU 指令。这也正是“人机交互”的基础：装置，以人可以理解和传递的形式，通过一定介质和中间转换，表达为确定的设备之间的交换数据。例如鼠标，人理解和传递的形式是移动与单击，鼠标作为中间的转换装备，最后将位、字节等数据通过设备之间的传输，送至计算系统的其他设施，例如端口。

从这个角度上来说，所谓的人机设备本质上仅仅是一个“数据采集”和“数据反馈”的终端，采集的是人的行为，反馈的则是人的知觉系统所能理解的信息（物理的、生理的、化学的、机械的、光学的等）。包括采集与反馈，以及该装备的内部构造与行为等，也是可以通过计算系统来控制的，那么它又将变成前面所述的、程序员所关注的两种“数据”。

五

我们在程序设计中表达一个数据的时候，总是在描述数据的某些侧面，而不是它的全部。其关键在于，我们事实上不会面对这一数据的全部方面。进而推论：我们既无法、也不必完整地表达现实系统（的种种“数据”）。例如我们说硬币的正反面，可以表达为 0、1，但这一项叙述只表明了 0、1 与硬币两面的映射信息，并没有实际表明：

□　硬币当前是正面亦或反面。

□　旋转中的硬币是正反未确定的。

□　硬币正面明确的称谓是 front？或是“正面”？或是 yes？又或“反面如何称谓”？

□　“硬币是正面的”在发生时，直到识别者得到这一结论的过程中，可有变化？

因此在我们约定了表达数据的规则——即数与数据之间的映射关系之后，还需要约定表达数据的方面。如前所述，我们不需要说明这个数据的全部性质，只需要探求一个最不可或缺的子集。综合目前在计算机语言及其抽象概念上的种种尝试，可确信包含于该子集中的有：

标识、值和确定性。

六

首先，我们需要一个标识符系统来“标识”所有我们要操作的数据（例如值与引用），以及这些数据的操作方法（例如运算符）。现实是：任何一个不被显式地或隐式地标识的数据，都不可能参与运算过程；任何一个不被标识的行为，都不可能在系统中执行操作(6)。

继续思考这些标识对我们而言是相当有意义的。例如说，我们用 aNum 标识了一个数据，请问这个数据是指 0 呢，还是指 1？或者我们再设问，即使我们用 aNum 标识了数据 1，请问这个数据现在究竟是 0，还是 1 呢？

上面这两个问题看来是文字游戏，但确实是计算机语言和编程中最核心的一些设问。其一，它涉及一个标识是否有其存在价值的问题。亦即是说，数据（包括其操作方法，亦作为数据）是否明确地作为该标识所表明的——内容，亦即数据，亦或者更明确地表述为计算机术语的“值”的——意义而存在(7)。以第一个问题为例，它表明：

这样的声明仅只是标识了该数据，但不存有值；而当我们使用下面的代码来声明时：

才表明了这个 aNum 是存有值的。

其二，它涉及计算环境如何认识上述值的问题。我们继续以第二个问题为例，上面的声明并不表明这个值是否存有变化的可能。因此下面的代码声明：

表明 aNum 是可能变成 1（以及其他的任何数值）；而以代码：

来声明时，就表明 aNum 是确定的，不可能变更为 1 或其他值。

也就是说，程序中有所谓的变量或常量之分，正是计算环境设问数据“确定与否”的种种侧象。

七

从计算机存储设备的角度来看，所谓的数据其实只是一个个单元格（cell）(8)，例如内存中的某几个位置。如果我们的数据“永远确定”，则意味着这些单元格要永远存留并且不可变化。这当然不可能，因为不可能有永远的或者无尽的存储。因此任何在计算系统中的数据都存有生存周期的问题，我们讨论的确定与不确定——或者你可以（暂时地）看成常量与变量——是以其生存周期为背景的。一旦离开这个背景，则标识仍可能存在，但讨论数据“确定与否”便不存在意义。

生存周期是一个数据的性质，亦或是运算系统的性质在数据上的投影？这是一个哲学化的命题。但现在，我们可以仅仅将它当成不同视角带来的理解差异。以我们当前所需要的——我的意思是数据的——视角，我们可以认为那仅仅是计算系统的性质。例如一个进程的生存周期，即是该进程中数据的生存周期——对于数据来说，生存周期只是其背景的、自然的规律。

因为生存周期是有限的，所以“数据确定”是可能的。例如我们以一个确定的空间（我们通常称为 Buffer，或 Cache，或特定内存块）来存放数据，那么我们可以在数据填满这个空间

之前保证任何数据的确定性。

八

但数据也可能是“不确定”的。分析我们上一项假设，它所包括的约束有二，其一是“确定的空间”，其二是“填满该空间之前”。当这些约束不被满足——数据所必须的单元格（cells）数量不足以表达我们的运算对象，且单元格背景上的生存周期不足以维持数据的确定——的时候，数据将是不确定的。

现实中，二者往往是同时出现的，例如主机接收来自端口的鼠标的位置数据时，我们既不能确保它必然在一个生存周期中得以处理，也不能保证有足够的空间来存放这些数据——从开机到关机。

因此数据即使被标识后是确定的，这种确定也必止于它从存储（这个背景）中消亡——例如数值 0 变成 1，则 0 即已消亡了，而之前它是确定的。从存储的角度来看，由于单元格（cells）有限，所以必然发生数据更替，进而导致上述消亡的出现。因此，数据的不确定性，首先是由存储背景的有限导致的，而并非来自于数据自身。

数据一旦存在“不确定”的可能，则计算系统的严密性就受到了挑战——计算的不确定性是对机器计算是否有价值的终极拷问。

第二节　逻辑

一

在计算系统中，数据基本上来说有两个方面的性质，一是指它的标识，二是指它的内容，亦即是值。出于计算环境的限制，在以“标识与值”这样的方式描述的数据上，也存有第三个方面的性质，即值可能是确定的，或不确定的。

既然确定性是数据与使用这些数据的计算系统的最终极的问题，那么我们权且认为数据是确定的。说“权且认为”，是因为我们晚一些会讨论到它“不确定”的一面，而且为了使之可以被整个计算系统接受，我们会试图将“不确定”作为“确定”的一种特例，由此使整个计算系统的行为确定，进而让我们的计算有意义。说“数据是确定的”，是指我们可以假定标识 aNum 从开始作为数据标识，一直到它失效，其内容都是一个确定值。

当我们确定“数”是什么的时候，才能确定地描述基于该数的“算”是什么。例如说我们确定了二进制数，因而确定了基于 0、1 的加减乘除等。正是数的抽象与算的规则这种紧密的绑定关系，决定了我们不能将这一“算”的规则应用于十进制数。但这并不是说，将正确的数与正确的算耦合在一起，就可以得到一个正确的计算系统。

在我很小很小的时候，邻居的大人总喜欢出一些数学题来考我。他们问：嗨，小子，你说说“三加二减五”等于多少啊，我回答：是零。然后他们又问，那“三加二乘以五”又等于多少啊。

我回答：是二十五。

然后大人们就开心地哈哈大笑。我在 3+2、5-5、5×5 几个运算中，都正确地得到了结果“数”，并正确地应用了基于十进制数的“算”的规则，但答案是错的。

二

现在我们都知道原因：在“三加二乘以五”中应该先计算“二乘以五”。对于 3+2×5 这个题目来说，单纯地做：

这样的计算是不行的，因为上面的解题中出现了“先”计算什么，与“后”计算什么的问题。

由此可见：无论是 3+2 还是 5×5 等，都是数值的计算；这些计算要正确地表述一个解题过程，还需要一个正确的逻辑描述，例如“先后”——即是指，我们要按某种顺序逻辑来应用“算”的规则。然而这样“正确的逻辑描述”有哪些呢？

这倒不需要我们再逐一列举，或像我一样回顾数学知识的点滴来源。Dijkstra（戴克斯特拉）(9)对这个问题有过非常严谨的数学论证，他指出：（我们）有三种思维方法用来理解一个程序，即枚举法、数学归纳法和抽象。除顺序逻辑之外，他指出枚举法、数学归纳法分别可以用程序中的分支逻辑和循环逻辑来表达(10)(11)(12)。例如，枚举法的基本思维是，对于一个条件集，

□ 如果条件 n 不成立，则条件 n+1 可能成立；若条件 n+1 仍不成立，则条件 n+1+1 可能成立……

□ 如此非此即彼，则当所有条件不成立时，集中没有成立条件；否则，

□ 条件之一成立，则集中有成立条件。

对于上述思维过程，就可以用分支逻辑（分支以及多重分支语句）来表达。

三

我们已经触及到计算系统的构成，以及在这样的计算系统上正确计算的诸多要素。一些看起来很明显的要素包括：数、数据（标识、值）和正确的逻辑描述，另一些不太明显的要素包括：算、思维方法和确定性。

Dijkstra 接下来用顺序逻辑统一了分支与循环逻辑，使之成为“顺序机器”上的最基本的正确的逻辑描述。他说明，我们可以从形式上将上述三种逻辑表示为图 2-1 所示的图形。

图 2-1　三种逻辑表示

很自然地可以发现：分支逻辑与循环逻辑其实都只有一个入口和一个出口，因此它们也自然可以作为顺序逻辑中的 S1…Sn，而不会破坏顺序逻辑的基本规则。更进一步，既然分支逻辑与循环逻辑是可被证明为正确的，并可以作为“顺序机器”本质所设定的顺序逻辑的一部分，那么由这些逻辑构成的“程序”也就必然是正确的。

这里的所谓正确，包括三个意思：一是程序能正确地描述人的思维；二是程序可以由机器正确地执行；三是机器执行的结果正确地符合人的思维的预期。不过这所有的“正确”仍然依赖两项前提：一是计算系统是一个“顺序机器”，二是在每一个用于计算的阶段（S1…Sn）中的数据，是确定的。

最后这一点——数据的确定性，正是顺序逻辑的必然结果：对于一个确定的逻辑而言，一个确定的输入，必有一个确定的输出。所谓输入与输出，若是数据，则在第一节中所述数据的内聚与外延的性质保障了这一结果；若是逻辑，则如上的形式化证明便保障了这一结果。

第三节　抽象

一

抽象是人们理解已知与未知事物的基本能力。例如你给旁边的同事甲介绍说：这段程序是张三写的啊。这时甲知道了“张三”，但并不知道张三的年龄身高、衣着打扮，所以这“张三”便是一个抽象。如果此时你把李四拉来旁边，说：不过这个人也出了些主意。这时，甲看到了活生生的，有年龄身高、衣着打扮的一个具象的人，却不知道这个人是李四。

抽象与具象是我们对事物的全部认识。只有当你指着那个人说“这是李四”的时候，同事甲才能把一个具象与抽象联系起来。所以事实是我们作为具象存在，而又用抽象来表明自己存在。这既构成了我们的人类世界，也同样构成了我们的计算世界。而这样的关系，在程序中不过是一行代码：

在此前的讨论中，我们说 aNum 是一个标识，上述代码声明了 aNum 的两项性质：它是变量，它指代数据 0。从抽象的标识 aNum，到它作为具象的上述两项性质，我们事实上已经看到

了（并非物理上的）计算系统的绝大部分构成。无怪乎 Dijkstra 说：“人们一旦了解在程序设计中如何使用变量，他就掌握了程序设计的精华。”

二

但是在这一行代码中，除了表达上述两项性质所必须的内容之外，还有一个“等号”（=）。在不同的语言中，这个等号可能存在两种含义(13)：其一，它是仅仅叙述 0 与 aNum 之间的指代关系；其二，它表示将 0 这个值存放到 aNum 这个标识所指示的存储（cells）。

一些语言中的等号同时包含上述两个含义，另一些却只有其中一个含义。而我们知道，一个标识——例如符号“=”仅仅是一个抽象。那么上述观点是说：语言在等号上的抽象含义并不相同。

拿这三种可能含义之一——“表示将 0 这个值存放到 aNum 这个标识所指示的存储（cell）”来讲，这是一个具体的行为，它表明计算机器的各个部件间配合完成的一个操作，即运算单元将一个数 0 置入存储单元。从人的角度来看，这类似于我们向机器（包括运算单元与存储单元）发送了一个命令“aNum = 0”，于是机器就执行了这一命令。

这就是“命令式”计算范式的基本抽象。

三

然而我们知道，“将数置于存储单元”这样的事情并不是计算所必须的——它只是“计算结果的表示法”。因此我们将这个部分从代码：

中抽去。则运算部分可以表示为：

这个“运算部分”——作为整体来理解——的具体内容有两种可能：对于计算来说，表明“数 0”的方法，就是一个 0，或将之表示为一个运算 f()的结果。设 f 用 equ0 来标识——既然“运算部分”整体可以理解为一个标识，那么我们也可以换一个标识来表示这个抽象：

既然上述 equ0 是一个计算过程，那我们可能通过无限的步骤来完成这一计算。亦即是说，equ0 仅仅关注该标识与其含义的确切关系，其具体指代为 return 0，或者是某个耗时无限的计算过程，但并不是这个抽象本身所关注的。

现在，我们提出最后一个推论。既然 equ0 可以指代无限的计算步骤，那么它必等价于图灵机的“顺序逻辑”中所有步骤；既然 equ0 可以指代图灵机所有的顺序步骤，则必然能指代顺序步骤的两种特例：分支与循环；既然 equ0 既可以指代计算的要素“数”，又可以指代

计算的要素“算”，还可以指代描述正确计算所必须的逻辑，那么 equ0 本身——在概念抽象上——必然等同于一个完整的计算系统。

这就是“函数式”计算范式的基本抽象。

四

关于函数，这里再做最后一点点补充。Dijkstra 说“在命名一个运算和使用一个运算之间也存在着一种抽象”，这里的“命名一个运算”即是函数的本意。

基于此，Dijkstra 提出在使用中只注意“（函数）做什么”而不必问“它如何做”。他强调这一过程与使用定理而不必问定理如何证明是一样的。他用这种“偷懒”的法子来证明：使用函数与使用定理一样可行，因而由函数构建起来的计算系统也有着如同用公理、定理构建起来的数学系统一样的正确性。

这一证明的关键假设是：本质上相同的抽象系统，其解集的抽象本质上也是相同的。而我们人类在自然科学领域中的全部知识，皆来自于这一假设的正确性。

————————————————————————

(1) 这里涉及 4 篇重要的学术论文，包括乔治·布尔的《逻辑的数学分析——关于演绎推理的一篇随笔》（1847 年）与《逻辑的演算》（1848 年），以及克劳德·艾尔伍德·香农的《继电器和开关电路的符号分析》（1938 年）和《通信的数学原理》（1948 年）。

(2) 引自《编码：隐匿在计算机软硬件背后的语言》，Charles Petzold 著。关于二进制数、布尔逻辑以及与此相关的电子学知识，可以参阅该书第 9～14 章。

(3) 数据是客观对象的表示；而信息则是数据内涵的意义，是数据的内容和解释。这里选择性地使用了这一释义，其原因在于“数据是表示”适宜作为对程序设计（特别是其中有关数据的表示法）的后续讨论的基本背景。

(4) 在 Pascal 语言中的一个常用函数，意为 increase()，递增 1。

(5) 数据的交互是可以表达为数据的，即在交互双方本质上仍然是通过数据来传递行为指示——“控制与被控制”是两个设备之间的行为关系，“数据”包含了行使行为所需的全部信息，而“（某种）行为”是设备自身的能力。

(6) 关于这一点是如何确立的，是我们在第五章第一节中一切讨论的起始。

(7) 这里说到“值”，是将数据标识的含义，与该含义的内容——一些具体的性质分开。例如说我们谈到“树”这个名词（或作英文的 tree 这个单词），则标识一个自然中存在的事物，但树的内容——例如高度、树龄等这些“值”并没有被叙述。而确定性，便是指“树”这个

标识是否包含了这些“值”的一个确定描述。

(8) 这一概念（Cells）引自 Peter Van Roy 的著作 Concepts, Techniques, and Models of Computer Programming。在本书下一篇的附录中，对这一概念的提出以及它与语言范型之间的关系作出了进一步的释义。

(9) 艾兹赫尔·戴克斯特拉（Edsger Wybe Dijkstra），荷兰计算机科学家，结构程序设计之父。1972 年图灵奖获得者。本书对 Dijkstra 的观点的引用，均出自《结构程序设计》第一篇，即“结构程序札记”。于此，后文中不再复述。

(10) Dijkstra 是用数学方法来证明分支与循环（和递归）逻辑的正确性，我们同样可以把这个证明过程看成一个映射关系，即上述逻辑可以视为思维方法在计算系统中的映射。

(11) Dijkstra 并没有用同样的方法来证明顺序逻辑，而顺序逻辑正是图灵机的本质设定。因此，在 Dijkstra 的论证中，他强调“只讨论关于‘顺序机器’的程序”。

(12) Dijkstra 提到了抽象，但没有对抽象在逻辑证明中的作用给出类似的数学证明——尽管他事实上在后面（不太明显地）给出了一个抽象逻辑证明的实例。

(13) 事实是可能存在无限种的含义。为了简化我们的讨论，我们这里只讨论常见的两种，即 aNum 与 0 之间存在赋值关系的情况。至于将“=”用作等值比较运算等类似情况，我们暂不讨论。

第五章　语言及其面临的系统

一个和尚挑水喝，是函数的功能问题；两个和尚抬水喝，是运算能力的分布问题；三个和尚没水喝，就是系统问题了。

第一节　语言

一

就程序设计语言来说，它涉及两个事物，其一是程序设计，这个词多少有些美化的成分在里

面，因为它原本的意思仅仅是编程（programming），而后来才被演译成了程序设计（program design）；其二是语言（language）。

什么是语言呢？解释这个问题跟说清“什么是我”一样地困难，因为如同“我在解释我”一样，我们也正是在“用语言来解释语言”。语言影响了我们全部的生活，是我们全部的知识：我们在口头上讲的，在书本上写的，以及在头脑中思维的，无一不是语言。但是大多数时候，我们只是“被学会了”一门语言，而并没有去认识它是什么。

我们口头上讲的、书面上写的以及头脑中形成映像的三个东西被统称为“语言”，那么显然，上述三个事物仅仅是“语言”在不同载体上的表现而已——用我们此前的一贯措辞，就是三者都不过是语言的不同侧面。因而真正决定了语言之为语言的，决不是书面上的字符，或口头的发音，或头脑中的那个意象。就某一门语言来说（例如汉语），其一，如果书面上的字符决定了它（是这一门语言），那么它无法包容（该语言的）古今文字的差异；其二，如果口头上的发音决定了它，那么它无法包容地方口音；其三，如果头脑中的意象决定了它，那么它无法包容任何还不存在的事物。

所以我们讨论的语言，是不能用其外在形式来定义的。通常我们对语言的定义，仅仅是说明它的功能，即语言是一种事物与事物之间沟通的工具(1)。由这个定义方式来看，程序设计语言，是指计算机——程序的使用者，与人——程序的定义者之间沟通的工具。

二

程序设计语言——这种工具有什么性质呢？或，究竟是什么决定了一种语言称为 Java，而另一种叫做 C#呢？它们之间存有何种不同，又存有哪些渊源呢？有趣的是，通过分析现有的种种程序设计语言，我们发现——正因为这些语言是我们人类自己创造的，所以——如同人类的自然语言一样，程序设计语言也总是有着三种基本性质：语法、语义与语用(2)。正是这三种性质，使得它区别与其他语言，而又能从其他语言那里有所借鉴以及沟通。

语法是指我们表达内容的形式。这一形式首先与不同的表达手段有关，例如同一个意思，我们的口头表达和书面表达是不同的。其次，即使表达手段相同，也会因为介质的材料性质存有差异而导致形式不同，例如钟鼎文和白话文都用于书写，但显然钟鼎文不能像白话文那样冗长。类似地，在我们的程序设计语言中，早期的程序输入就是电子开关的开合，因此代码会是一些操作命令，而现在我们可以将之输入为接近自然语言的程序文本；早期的运行环境限制要求代码必须尽量精少，而现在我们则考虑通过规整而冗长的代码来降低工程整体的复杂性。所以，语法是对语言的表达手段，以及对表达的条件限制加以综合考虑而设定的一种形式化的规则。

语义是指我们表达内容的逻辑含义。语义有两项基本性质：一，必须是一个含义(3)；二，该含义必须能够以某种基本的逻辑形式予以阐述。语义还有一项非必须的性质，即，三，上述的逻辑所表达的含义可以为语言的接受者所知。

略为讨论一下第二项性质。为何语义必须可以阐述为一种基本逻辑呢？因为语义定义为内容的含义，而这种含义可以由多种形式来表现，因此如果它不能用一种基本逻辑来表达，也就没有办法在多种表现形式之间对它互作验证。例如不能用书写的方式来确定口头转述的正确

性，或反之也不能通过口传心授来传播书本知识。自然语言中的这种性质（部分地）可以由基本逻辑的矛盾律来约束，即“一概念不能既是该概念，而又非该概念”。正是我们的文字记录与对话交流等内容中存在着这样的一些基本逻辑，所以它才可能科学、严谨以及正确。

第三项性质对自然语言来说是非必须的——如果一个人自言自语，那么他的言语可能仍然是有语义的，只是这语义不为他人所知。但这一点对于程序设计语言来说却是必须的，因为我们设计这样一门语言的目的，正是要让我们所要表达的含义为计算机所知。而正是这第三项性质，加入了对“语言的接受者的理解能力”的限制。出于语义的前两项基本性质，这种理解能力也必然由两个方面构成，一是指含义，二是指逻辑。

我们回到了上一章所讨论的内容：计算系统的要素，包括数、数据和逻辑，以及在此基础上进行正确计算的方法的抽象，即计算范式。只有通过这些组织起来的语义，才可能被（我们此前所述的）计算系统理解。这些语义与其表现形式（即语法）是无关的，有其基本逻辑存在。

我们所谓的“会编程”是指对这种语义的理解，而非对某种语法的熟悉。正因如此，我们才可以在 Java 上实现某个程序，又在 C#上同样实现它，在（使用这些语言的）类似的仿制过程中(4)，

不变的是语义。

三

我们来稍稍讨论一下语用的问题。

从对自然语言的观察来讲，同一句话——即语法和语义都严格相同——在不同的场合（语境）中出现，却可能有微妙的甚至是迥异的差别。讨论“差别”这个问题时，我们要先将语法从中别开，例如：

“这难道不是吗”与“难道，这不是吗”

在后一种表达中用语法带来的强调效果；也要将语义从中别开，例如我们使用：

来表达算术，但也可以用它来引申为人与人的合作，这两种语义都是确切且又不同的。最后，我们还需要将某些语言因其不严谨以及使用习惯所导致的歧义从中别开，例如：

早起的鸟儿有虫吃

既可以理解为“有虫吃鸟”，也可以理解为“鸟能吃到虫”。

在这几种情况区别开之后(5)，语用讨论的是语言背景的因素，例如：

“去死!”

用在战场中，表示愤怒、诅咒与呐喊；而在情人间即使连标点都不变，也可以表达亲昵。这在语义的组织与逻辑上，以及在语法的构造与表述上都没有任何的不同，但因为场合而含义有别的情况，是语用的问题。

显然，如同我们此前所说的“计算的不确定性是对机器计算是否有价值的终极拷问”，我们并不希望在使用一种语言与计算机沟通的时候表达出上述的不确定的含义，或者反过来，计算机给出我们一个不确定的结果。因此事实上我们在设计计算机（软件与硬件）系统之初，就在尽力避免与之沟通时存在的语用问题。亦即是说，在严格的计算系统中，语用——这一语言的背景因素被限制在计算机的初始环境中，从而使“语义+语法”能够描述确定的计算及其结果成为可能。

但是在计算机的应用中，领域特定语言（DSL，Domain Specific Languages）其实是基于对语用学的研究与实践。所谓领域特定，即重设了“严格的计算系统”这样的背景。所以在这类语言中，我们可能看到与此前讨论的“计算系统的要素”不同的内容与逻辑。但是从语言的性质来看，它仍然是基于语法和语义，并且限定语用（领域环境）的。

四

语法与语义是语言的两个基本性质，分别指代语言的两个方面：形式与内容。就经验来说，可以想见的：形式与内容不一致——亦即是所谓的“辞不达义”的情况，就必然会出现。在现实中，我们可以通过对同一事物反复地(6)、从不同侧面(7)和用不同方法(8)描述来解决“辞穷”的困境。而我们显然不可能在程序设计中这样做，因为计算机对事物的理解形式很单一，此其一。

其二则是我们不必这样做。因为此前我们讨论过，计算机的理解能力是有限的，只包括数、数据和逻辑以及在此基础上进行正确计算的抽象，所以我们只需要约定语法与这些计算机理解能力范围内的东西之间的唯一关系，那么计算机所理解的东西与我们描述的东西，就有了唯一的映射关系。不过换一个角度来看，我们也必须按照这样的约定来设计我们的语法，使之唯一对应一种计算机理解能力范围内的语义。这在程序设计的术语中，叫做绑定。

五

绑定有很多种情况，几乎所有在程序代码中用到的标识，都存有绑定的问题。最常见的例如变量：

上述一句代码（的语法），对应着三个语义：

□ 有一个标识，记为 aNum；

□ 标识所指代的数据，其值是可变的（即，变量）；

□ 该数据当前值为 100。

对于大多数语言来说，第二个语义是一项执行过程中的限制，亦即表明任何可以访问到 aNum 标识的代码都可以改变它的值。为了简便，我们暂不讨论这一个语义的绑定问题。但第一、三个语义所述的：

□ 有标识 aNum，其数据当前值为 100

就明显作用于我们的计算：如果数据无值，或值不确定，则我们无法进行后续的计算过程。所以

在语义上的“值 100”将在何时被绑定到标识 aNum，就是一个相当重要的问题。

以一个代码片段来看（相信我，这是实际可运行的代码）：

根据我们对这段代码的语法约定，第 2、5 行决定了代码片段中的标识符的生存周期(9)。但在这个生存周期中，aNum 是何时有“值 100”的含义的呢？

对于这个问题，一些语言认为，表达“aNum 变量具有初值 100”这样的语义，应该是计算过程的前设，即对计算前提条件的声明。而声明并不是计算，因此声明语法与执行语法也应当分别对待。例如 Pascal，就处理为这样的语法（在 var 部分声明，在 begin…end 部分执行）：

另一些语言则认为，声明语法可以理解为对执行环境的预置，也有执行含义，因此可以允许“即用即声明”，例如 C。但是语义上，这是因“（需）即用”而进行的声明，不可能出现“已用”而未声明的情况。所以即使 C 语言，也会因为语义上无法解释，而判断上述代码 1 违例。

还有一些语言认为：

是一个语义，它只表明标识 aNum 的存在，直到函数调用时才绑定另外一个语义：初始操作，即为函数内所有存在的标识进行初始化。在这样的认识下，“代码 1”中的

就具有了完整的语义。这个语义（即“初始操作”）与实际的计算机行为，是在把

process_part_one()作为函数调用时，才进行绑定和实施的。其效果是：

□　在生命周期（代码 3 的 2～4 行，代码 1 的 2～6 行）中，记有一个标识 aNum；且，

□　在该标识所处生命周期开始时，总会有一个初始化动作，使该标识具有一个初值：无值；且，

□　设整个计算环境中，无值是一种值，记为 undefined。

你可能已经知道，这就是 JavaScript 的实现方法(10)。进一步地，除开上述代码在生命周期上的解释之外，代码 1 被理解为：

我们看到，代码 1 中的语法：

所具有的两个语义（本小节开始处的语义一、三）被分别绑定在函数调用开始和代码 4 的第 4 行；且一个标识的初值含义，总是被确定地约定为 undefined。由此一来，上述代码 1 整体的语义就确定了。

六

我们所谓的会“编程”是指：将我们的意图表达为计算系统的理解能力范围内的语义。而这种语义：

□　由计算系统与程序员共同确知的数据与逻辑构成；且，

□　最终可以由某种计算方法在指定计算系统上实施以得到计算结果。

这里的计算方法并不是指“算法”，而是指对某种计算实施过程的抽象，例如在上一章第三节中所讨论到“命令式”和“函数式”这两种计算范式。所以，

会编程与掌握某种语言的语法形式是无关的。

编程实质上是一种在语义描述上的能力修养。具备这种能力之后，语法也就无非是一些规则、限制和对不同计算系统的理解能力上的差别了。所以“计算机程序设计”这门功课应该教你编程，而不是教你使用一门具体的语言——我们现在大多把它当成语言课，实在是本末倒置了。

第二节　从功能到系统

一

我一直有一个问题：为什么 JavaScript 不适合大型开发？更进一步的问题是，JavaScript 可以应付何种规模的开发？或者，我们泛义一点来讨论这个问题：语法与语义上的特性，是否决定了该语言可能适宜的开发规模？

对于这一系列的问题，我希望尝试通过对第三个问题的分析，回溯至对第一个问题的探讨。首先我们需要界定“开发规模”，而这缺乏一个必要的、科学的分类方法。以我的经验来说，我用四个显著的关键词来区分我们面临的开发规模的不同等级，分别是：功能（function）、程序（program）、应用（application）和系统（system），如图 2-2 所示。

图 2-2　开发规模的不同等级

并不是说一段代码中具有了上述某个级别的关键词就有相应的规模，而只能说这种语言具有对这个级别的开发规模的支持。具体语言对这个级别可能支持得好或不好，但“好”的程度并不是我们讨论的问题，因为这是一个变化的、发展相关的状态。

我们需要另外说一下项目（project）这个关键词。“项目”主要是一个开发过程中的、组织上的用词。就其组织的形式而言，更进一步地会有项目组（project group），退一步则可能是一个活动或事件（action/event）。但是，这些都与我们这里讨论的开发规模无关。你可能为一个只有三行代码的微型程序创建一个项目，或者一个项目组中仅仅只有三个空程序，这些说明不了你的开发规模，而只是你对后续开发活动的组织形式的设定。

所以“语言具有什么样的特性”出自语言设计视角，并对“项目组织成多大”有支持作用，但不是后者的全部。那么接下来的问题还有两个，一是“为什么存在支持作用”？二是“哪些特性，有何种的支持作用”？

二

我何以将开发规模分为上述四种等级呢？因为这四种等级随计算机应用的演进历史发展而来，具有各自不同的特性和表现。

功能（function）是计算机的本质能力，它包括最初的计算能力，以及计算能力在数学抽象上的、最大粒度的表达。这类语言的目仅仅是“完成计算”，其特性集中在对计算机的计算能力的抽象和实现上。一般来说，一门语言只要具有：

□　数（number）的定义

□ 数据（data）的定义

□ 计算能力（formula/expression）

□ 逻辑能力（顺序、分支与循环）

□ 一个计算过程的约定（命令式、函数式或其他计算范式）

那么它就已经是一门这样的语言了。这在个级别上，最大规模以引入函数这一抽象概念为止（即具有将一系列的计算过程定义为函数的能力）。因为函数等同于“数学定理”，所以他可以像一个公式或运算元一样直接使用。而这样的语言被创生的目的，通常是通过计算能力（包括函数）与逻辑能力，对一系列的数——亦即数据进行计算。因此可以将之定义为：

即，“计算范式+计算能力（数据）”：

出于表达的需要，我将这一等级上的语言统称为“计算语言”（computing language）。需要注意的是，一般来说计算语言通常不适合应用软件开发，而只是数学与计算机科学在应用于跨学科研究领域中的工具。

接下来三种等级以计算语言为源起，是语言设计在结构化这种思想下的、两种方向上的发展结果，如图 2-3 所示。

图 2-3 “应对开发规模”的结构化求解：四种等级的本质

三

第一种结构化方向是指：通过在计算要素上的结构化来增强计算能力。在这一方向上的努力，将我们的开发规模推进到程序（program）这一等级。我所见过的，对于程序的最初、最正确和最直观的定义为：

即，“算法+数据结构=程序”(11)：

但是这里的结构（structure）主要作用于数据（data），所描述的是对算法（algorithms）中所用数据的抽象。

但是在更大的计算规模中，计算对象——数据的复杂性固然是一个问题，而计算本身的复杂性也是一个问题。因此对于 a+s(d)这个定义来说，Dijkstra 在《结构程序设计》中的描述更为完善，即：

显而易见，后者用计算的结构化 s（f）来替代了算法 a。原意为(12)：对于一个计算过程，（通过提炼）抽去每次计算所处理的特殊值，余下的不变的东西就是这个“算法”自身。由于对过程中的每一步都进行了“动态提炼（算法）和静态提炼（数据结构）”，因此这里的提炼结果 s（结构）在计算过程 f 和数据 d 上是匹配的、共生的，即

其三，程序代码本身在组织上也还存在复杂性的问题。对于这第三个问题，Dijkstra 讨论到“组织和编排程序”。不过在具体背景之下，他所讨论的仅仅是对“函数/过程”(13) 进行结构化地组织与编排。

这里需要明确一下在该背景下的函数这一概念。在上一小节的计算语言中，函数是与数学公理类同的、面向“数”或“数据”的一个计算过程。而在本小节所讨论的程序（program）这个规模之下（以及该规模下的语言之中），函数是算法或算法中的一个子步骤的代名词。在 program 这一语境下的函数，与确定的结构有关，既包括计算逻辑的结构，也包括计算数据的结构，是

中三者合一的概念。如同《结构程序设计》用“珍珠”和“项链”来做的比喻一样，从珍珠的角度来看，是可以在整个程序中被替代、被优选的(14)；从项链的角度来看，可以是暂时不完整的(15)；从串起项链这件事上来看，顺序是确定的(16)。

我将这一等级上的语言统称为“编程语言”（programming languages）。我使用这个具有歧义的称谓的原因在于(17)：这一类语言仍然是在计算要素上加以增强——尽管对于函数来说也存在着代码的组织与编排上的增强，但归根结底是针对计算要素的，并且是在该抽象上的自然延伸。

在这一阶段中，计算机主要在专业领域中使用，计算量的增加是程序规模变化的主要原因，同时又要求对目标系统有足够的抽象表达能力，因此在数据抽象程度变高的同时仍然强调语言具有完备的计算能力。我们可以将操作系统、网络系统、通信调度、分布式运算环境等绝大多数基础系统的核心部分，以及其中大量利用系统特性的工具软件，都归于这类语言所面向的开发规模。其主要特点是：大量的计算、数据抽象与计算环境相关、主要算法可以在不同的软硬件环境中通用或重现、用户主要入口是操作系统的约定或直接的硬件系统界面。

四

在第三个问题上的尝试，推动了结构化思想在另一方向上的发展(18)，即通过在程序组织上的结构化来解决规模问题。这一方向上的主要动力，是程序所应对的需求渐渐扩展到非专业领域，用户的入口变得多样，甚至同一功能涉及到多种用户/角色的、不同时间与时序上的参与。这些也是软件开发工作的成果从专属程序进化到软件产品的主要标志。

我所观察到的第一个与此相关的语法元素是单元（unit）(19)。因为一段代码可以写得任意长，也可以拆成几个单元，所以这些单元存在与否，就与这段代码要表达的计算功能是完全无关的了——既不是对运算的增强，也不是对数据的增强。接下来，我们发现一些单元可以对许多不同的 function/program 的开发提供支持，由此库（library）的概念就形成了。“库”用于集中管理一些具有相同应用范围、处理相关数据、解决相似问题的单元以及单元对外部声明的界面（interface）。如图 2-4 所示，这些语法元素之间的关系非常简单。

图 2-4　单元、库与界面间的关系

大多数情况下，库与库之间的区别仅在于功能集不同、应用范围不同、接口方式不同，以及我们经常关注的效率等的不同。（不考虑这些外在的表现）这些库内在的抽象思想与应用价值是完全相当的。

库的出现带来了一些组织程序的策略，例如静态链接库（Lib，Static Link Library）、动态链接库（DLL，Dynamic Link Library）、类库（Class Library）；又例如跨语言的组件库（COM 库、Component 或 Component Library）、公共程序集（Common Assembely）；再例如，远程/跨平台对象（Remoting Object，或 XPCOM）等。这些策略被种种语言实现为不同的（语言内置的）关键字，并由此决定了各自不同的一套支持语法。

所有这一切的目的，仅仅是因为那个最原始的发现：总有一些代码与当前处理的业务没有实际关系，可以被隔离到其他代码块（例如 Unit）中去。既然从组织策略来看这种隔离必然会发生，那么我们关注的问题就仅仅是上图的 Library 对外表达的形式，而非 Unit（或这一类的语法元素）的实际实现(20)。

这些变化既有语言进化的内在动力，也有软件规模变化这一外因的推动。当计算机不再仅是用于计算的工具，而变成软件产品的运行平台时，整个软件——传统意义上的程序（program）——的构成已经发生了明显的变化。现在，它必须面临的问题包括三个方面，如图 2-5 所示。

图 2-5　面临的问题与背景：Application 在“结构化”上选择不同方向的原因

也就是说，即使程序——我们仍然可以用第二个等级中的“程序”（program）来指代上图中的一个方面——在功能上的规模没有任何变化，那么也将面临如下两类需求：

□　使用人群由专业人员变成普通用户，将导致提出大量的非功能需求，即它作为“软件”的使用问题；

□ 由于使用人群的变化，维护和发布的工作对程序员变得不可控，因此将导致提出大量的非当前需求，即它作为“产品”的生命周期问题。

这二者，也即是“软件产品”之于“程序、功能”的区别。举例来说，在最初的操作系统中，我们开出一个内存块来使用。

□ 首先，我们仅仅是需要在别的计算过程中使用到它，因此它被实现为一个功能（function）(21)；

□ 接下来，操作计算机的专业人士认为，其实可以从内存中拿出一块来并将它 mount 成一个虚拟的磁盘，于是将它写成了一个“简单的”程序（program）(22)；

□ 之后，某个并不太懂计算机的桌面用户也需要上述功能，但受限于他的计算机操作水平、实际的应用环境(23)，他必然会在上述程序、功能之外，提出前述两类需求。

对于这类（软件产品的）用户，我们称支持非功能需求与非当前需求的程序为应用（application），而将这一等级上的语言统称为“应用程序设计语言”（application programming language）。这些需求促使一个应用中包括了多个领域的产品功能，一些显而易见的内容包括：

□ 与产品使用相关的界面，例如 3D 操作、游戏界面、数字监控仪表盘等；

□ 与应用环境相关的业务逻辑规则，例如企业组织架构、工作流、金融、期货等；

□ 与操作平台相关的开发接口，例如 ATL、Android SDK、GTK+等；

□ 与发布、维护相关的支持，例如帮助文档、安装包、工程文件管理、代码文档化等；

□ ……

这些内容的领域性特点使得任何一个或一类程序员都无法完全胜任它们的开发工作。单元等类似手段以及模块划分时需要隐藏内部信息的思想，都是在这样的背景下产生的。其产生的必然性，是泛计算领域本身的纵向隔离的特性所决定的(24)。也就是说，既然这是软件需求从专业计算领域走向泛计算领域(25)带来的规模问题，那么也要求程序在组织上的抽象必须能表达和匹配后者在领域问题上的纵向切分。

五

另一个带来崭新思考空间的语法元素是服务（service）。一个服务的发布、运行、使用、受益、检测与维护等整个过程都可能由不同的用户/角色来参与。例如，表 2-2 比较了生活中的邮寄与软件开发中的会话这样两个“服务”。

表 2-2　服务：比较生活中的邮寄与软件开发中的会话

* 这里假设会话服务提供过程中，所有的服务、业务等都有操作人员，即使部分可以作为自动化服务提供，我们也可以称之为干系人。

服务在操作系统及其应用软件中也有着重要的位置。以一个典型下载软件为例，它可能提供一个后台传输的服务（backTrans），那么该服务在 Windows 中的提供模式可能如表 2-3 所示。

表 2-3　服务：后台传输服务在 Windows 中的提供模式

* 这里假设这是一个桌面用户可选的服务，当不使用该服务时，基本的下载功能不受影响。

而当一个与此功能等价的服务通过网络来提供时（以 Google 账户同步功能在 Gmail 与 Android 通讯簿功能中的使用为参考），那么上述的模式可能改变为表 2-4 所示的结果。

表 2-4　服务：与上述服务等价的服务在网络环境下的提供模式

综观上述这样的一个软件需求与实现的模式(26)，是不可能仅仅以 Unit 以及更高的形式（库/套件）来提供支持的，因为其需求的本质在于“异地实现”。在这个需求下，由于“服务、功能部件，以及功能部件的操作者”这一系列行为完全不可预期，因此服务的调用方已经不能对实现者的语言与运行的环境作出任何限制。在这个级别上的问题，最终总是被归结为两个解决思路：

（1）交互界面是否可以表达为可实现的规则集；

（2）输出是否可以表达为可计算的数据项。

而简化该问题规模的方法也由这两个经验得出，即尽可能简单的界面规则与数据表示，例如 REST 和 JSON(27)。

对于这类需求模式，我们将提供服务集成能力以支持非本地需求的应用称为系统（system），并将这一等级上的语言统称为“系统程序设计语言”（system programming languages）。服务的提供能力与其所支持的层次，成为这个级别的语言特性的主要发展方向。由于服务所在的用户领域有着种种差异，因此在这个级别上的语言也需要提供特定领域的部署、维护与交互界面等特性(28)，例如 Java 中的 Beans、Annotations，或 Erlang 中的 Node、Port 等。

六

从计算机应用的历史来看，我们在语言中加入新的元素，其本质的原因正是旧的语言特性在应对规模（而非仅仅是计算）的时候显得力不从心，尤其是在应用与系统这两个规模级别中，（在语法与语义上的）语言特性体现出来的代码组织能力，相当大的程度上决定了这门语言所适用的开发规模。

最后，我们用表 2-5 总结一下不同视角对这些规模的认识。

表 2-5　不同视角对四种开发规模的认识

③ 使用这个有争议的名字原因在于，（基于一些历史的、习惯性的因素地来看）程序、应用和系统被我们统称为软件，而“软件工程师”是其实现角色的基本要求。当然，在此前，他还必须是一个程序员。

在继续讨论之前，我们先来明确一下将要讨论的对象。

首先，我们要讨论编程（programming），一些场合中也被称为程序设计（program design）。可惜在不太严格的场合下，几乎所有编写软件的活动都被称为程序设计——这是相当可怕的事实。因此，必须明确地对它做一个定义，即我们用程序设计来特指在功能与程序这两个级别上编写软件。相应地，我们将其后的两个规模分别称为应用开发与系统构建，这也将是我们后续要讨论的话题。

这几类活动有着显著的区别。程序设计主要应对计算要素问题，产出是一般含义上的程序；应用开发主要应对程序组织问题，产出是有产品化概念的软件；系统构建主要应对跨领域问题，产出是可持续进化的系统——例如平台，如图 2-6 所示。

图 2-6　两种“结构化”方向的本质差异

————————————————————

(1) 又例如，古文中对语言的定义是自言为言，与别人说则为语，进而有食不语、寝不言之说。

(2) 莫里斯（C. W. Morris）在他于 1938 年出版的《符号理论基础》一书中，最早将语法学（syntaotics 或 syntax）、语义学（semantics）、语用学（pragmatics）明确地作为符号学的三个分支。从而构成了现代语言学研究的三个主要平面。

(3) 概念或实体。即，在表达者的意识中需要表达的对象。

(4) 这也意味着，语义上的表达能力决定了一门语言是否真正有别于其他语言。语义能力上等价的语言，除了开发人员的喜好或运行平台的限制之外，所谓有益的价值仅是开发库的丰富与社区的活跃等。而所有这些，都是与语言的本质无关的。

(5) 这事实上也意味着计算机语言需要：语法明确，无情调修饰、无语义引申、无歧义。

(6) 例如沟通中的“再重申一下我的主张”。

(7) 例如沟通中的“我们换个角度来看这个问题”。

(8) 例如沟通中的“给你画个草图如何”。

(9) 如你所知的，这是一个函数。大多数语言的函数，都约定了其（形式上的）函数体内的标识符的生存周期。

(10) 在 JavaScript 的实现中，这里被实现为闭包。当一个闭包被创建时，它的上下文自然被初始化，而非（显示地）进行一个赋以初值的操作。你可以认为闭包是一个内存块，它创造的时候就是一块空（full by Nil）的内存。

(11) 这个著名的论断出自尼古拉斯•沃斯（Niklaus Wirth）。沃斯是 Pascal 语言之父，1984 年图灵奖获得者。《Algorithms + Data Structures = Programs》是他最有名的一本著作。

(12) 原文是“考虑一个算法和它所能引起的各种计算：从这些计算开始，抽去……”。如下一个脚注所谈到的，这里的计算过程是指 procedure，现在则常称为 function，也因此这里使用 s（f）这个描述形式。

(13) 在 Dijkstra 的叙述中，函数是偏向数学计算、有计算结果的；过程只是一段计算机处理，并不表明有计算数据返回。现在我们通常用函数来概含了二者，以 C 语言为例，过程是指返回值声明为 void 的函数。循此惯例，此后的讨论中，将仅仅使用函数这一个名词。

(14) “程序比较（程序验证、测试）”以及渐进优化的思想基础。

(15) “自顶向下”的结构化设计的思想基础。

(16) “架构”以及框架、流程等产出的思想基础。

(17) 另外的原因在于，对于多数开发人员来说，在这个级别上的开发才是他们心目中的、（被

教育所圈定的）计算机科学领域中的“编程”。以其后的例子来说，“写出一个操作系统”可能是许多开发人员心中的巨作。

(18) 我并不确定 Dijkstra 所述的“组织和编排程序”究竟在何种程度上影响了后两种等级的语言出现，但我确信这种思想是导致它们出现的原因之一。

(19) 在具体语言的实现上，它也可能被称为模块（module）等等，例如 Erlang。

(20) 这一观点的得来决非易事。这是 20 世纪 70 年代多个程序设计流派之间的主要争端，David Parnas 的“信息隐藏”的观点取得了最终的胜利，即模块内部的数据与过程，应该对不需要了解的信息予以隐藏。Brooks 最初持不同的观点，但在 1995 年的《人月神话》二十周年版中，他坦承“关于信息隐藏的观点，David Parnas 是对的，我是错的”。

(21) 这个功能基本上可以概括为：基于存储地址、用于限制访问边界、纯粹的线性计算（分配）的过程。

(22) 这可能是一个有 150 多个可选配置项的命令行程序，除了—help 参数就再也没有其他任何称得上文档的东西。

(23) 作为产品的用户，他还期望在个人需要与应用环境都发生变化时，仅支付少量的代价即可应对，而非重新购置。

(24) 最简单而又直白的说法就是：隔行如隔山。

(25) 但凡一切可以抽象为可计算对象，并通过计算系统来解决问题的领域。

(26) 我们可以将能在产品最终用户的环境中实现的需求称为本地需求，将不能在该环境中实现的称为非本地需求。

(27) REST（Representational State Transfer，表述性状态转移）是一种面向远程服务提供的架构方法，JSON（JavaScript Object Notation，JavaScript 对象表示法）是一种数据交换语言/规格。从这个角度来看，SOAP 与 XML 并不是复杂的方案。

(28) 这里讨论的是语言，因此限定这些特性是通过语法元素来实现的。尽管在开发包中，用第三方工具来提供支持也是解决这一问题的通常手段，但并不是我们主要讨论的话题。

第六章

程序设计的核心思想

但凡与计算机有些关系的领域中，总会有人通过编写一些代码、脚本或批处理来解决问题。这些工作大体上也包含了我们此前讲到的种种计算要素。但是这些只能算作编程，它并不等同于我们在这里谈论的程序设计，它是后者的一个初级阶段。学院式的编程是一种狭义的、升级版本的编码工作，而作为一个软件工程师，程序设计才是我们的必备技能(1)。

奠定我们如今主要的程序设计方法的那些基础理论和思想，在上个世纪的七十年代就已经成熟了。也正是从那之后，程序设计才从编码工作中脱离出来，成为一个有意义的、独立的名词。接下来的讨论以 Dijkstra 的论述为基点，包括：“计算的结构化 s（f）”和“数据的结构化 s(d)”两个方面，并且二者存在伴生的关系。

我们应该已经知道：抽象含义上的数是没有类型的。当我们讨论某种数的时候，即是在讨论某一类型的数，而非数的全部。而我们之所以分出这个数的型别，是指该类数具有相同的性质，例如整数可以由 1 不断复合。同样，数据既然是数的系列，则数据的型别（即数据类型，Data Types）可以理解为系列性质的不同，亦即通过何种系列的抽象来分类数据。何种系列的抽象，即是数据结构(2)(3)。

第一节　数据结构：顺序存储

一

所谓“排排座，吃果果，冬冬不在留一个”的故事大概是这样的：小朋友们排成一排或者许多排，然后老师给每个人发一个水果。之所以要先排排座，既是教大家规矩，也是避免老师发漏了或者发重了。大体上，我们的记忆里总会有这么一两个吃果果的场景——老实说，我现在看起来是在讲一个相当无聊的故事。

如果有一个有限大的空间用来放数据，我们能不能也将该空间规划成座位，然后将数据分发上去，从而使得每个局部空间上都有东西（或者还没有，就“留一个”）？这个问题之所以有讨论的必要，是因为只有我们将所有要处理的数据信息都放到计算机可以识别的环境中，计算机才能开始履行我们要它做的事情。

总的来说，我是在讲一个乏味的故事。但在计算机本质的抽象上，的确就是如此乏味。更为“生动”一点的叙述大概是这样：（1）我们要将所有的数据顺序地放到空间里去；（2）我们要考虑局部空间上有与没有数据这两种情况。

二

冯·诺依曼体系的计算机，是以控制器、运算器和存储器为核心的，分别映射我们此前讨论

的计算（范式）、算和数这三个方面。因此，如同我们对“算数”的理解一样(4)，存储器的结构以及它在计算系统中的抽象，也直接限制了计算及其逻辑。对于这个“抽象”，我们简单地说就是：顺序地址存储。

除了 Bit、Byte、Word、DWord 等与位宽直接相关的、具有数学或物理传输含义的类型——它们显然是顺序表示的值——之外，我们常见的在计算机中用的信息/数据有哪几种形式呢？布尔值算一种，它表示开关、正误等判断，是我们执行逻辑的一个基础；数值算一种，作为数学运算的对象，与我们计算系统的本质设定有关；字符（以及字符串）算是一种，因为它是我们程序员或用户可以正常理解的东西。这三类数据，构成了计算系统向（可用的）计算机发展的主要条件(5)。而这三种数据中，除了布尔值可以由计算系统的最小存储单元（bit）来表示之外，其他两种都必然面临“如何存储”的问题。

对于这个问题，“顺序地址存储”的核心思想是：设运算器可以通过一个（数值标识的）地址，从存储中获得一个（有限大小的）区域，则该区域可以表示为一个待运算的数据。以 Intel 系统的个人计算机（PC，Personal Computer）中，x86 芯片（CPU，运算器）为例(6)：

□ 芯片通过寄存器来读这个地址的值，因此寄存器可以表示的数值大小，决定了地址值的大小；

□ 同上，因为芯片也使用寄存器存储计算所需数据，因此上述区域（连续可读取的位数）的大小也是由寄存器可以表示的数值大小来决定的。

那么，首先，待运算的数据的大小就被转义成了区域的大小，进而变成了运算器与存储器之间的位宽问题。所以 32 位的机器上一个（能直接参与 CPU 运算的）值的大小，就是 232，如果要理解为“有符号整型数”则是±231，因为符号用掉了一个位（bit）。同样，我们要运算浮点数的话，也要把它“挤”在这 32 位中去表示，例如常用的 IEEE 754 浮点数表示法(7)。再者，我们要表示字符的话，也是通过对 32 位的组合序列做出定义，将它们一一表示为我们的书写字符，例如 ASCII 字符集、GB2312 字符集，以及 UTF8、UTF32 字符集等。

接下来，可能表示的地址，也被转义为上述的位宽问题。因为地址是数值标识的(8)，所以它表达的是一个连续空间中的位置（可以想象为数轴上有限区间的点）。根据上述的原则，以 32 位的 CPU 为例，该“顺序地址存储”方案：

□ 可以找到的最后的数据的位置索引就是 232-1，

□ 可以从该位置（以及其他任意位置上）读取的最大可能值是一个无符号的 32 位整型数（DWORD）(9)。

所以跳开我们熟悉的 x86 芯片去做软件开发的时候（例如单片机），我们常常需要问：寻址空间有多大，基本数据单元是多大。如果没有这样的信息，就无法通过某种语言编程去控制它，因为连一个基本的、与计算机交流的环境都搭建不起来。

三

上述这些被我们的语言称为“基础数据类型”。这包括两个部分，一是与位宽相关的，例如Byte，它是计算系统的直接映射；二是与应用环境相关的，例如Char，它与该系统对外的表示有关(10)。它们事实上本身就是对数据的结构化表达，最明显的就是浮点数的组合型表示方式，又例如字符的顺序型表示方式。但总的来说，这些基础数据类型，总是能“挤”在一个最大位宽的表示单元中去。

换而言之，在连续空间中表示完整含义。

然而就最初的需求来说，所谓完整含义是我们对数据而不仅仅是对于数的假设。例如，下面的数据是有完整含义的：

Hello World!

但这个数据如何在连续空间中表达呢？从自然形式的理解来看，它是12个字符。因此即使以每8位表示一个字符而言，它需要96个位。对于“顺序地址存储”来说，在32/64位机器中，即使能通过一个地址来找到它，也无法一次将它全部取到CPU中去运算。但是，它本身又的确是连续而完整的，我们不能孤立地从其中的一个部分来理解它。由此产生了一个问题：我们——程序员以及计算环境——该如何理解超过“（有限大小的）区域”的数据呢？

答案是：有一个起始地址的连续区域。

这里的起始地址与此前的“顺序地址存储”中的地址是同一概念，但“连续”性也必须体现为数值才能为计算系统所理解。因此对于这样的数据，我们需要两个数值来定义：地址、（连续的）长度。现在，由于有了“长度”的概念，因此我们通过“顺序地址存储”可能得到的数据就有了非常大的变化：

□ 其一，可以找到的最后的数据的位置索引不变，仍然是232－1；

□ 其二，可以从该位置识别的最大可能值是一个0…232-1长度的连续区域(11)。

或许你现在已经想到了“指针”（pointer）？不，我们现在还没有讨论到它。我们仅仅在讨论一个连续的存储结构，例如数组、字符串或结构体。

四

数组（array type）指的是包括某种相同数据（数组元素的类型相同）的连续空间(12)，它是顺序地址存储这一概念的自然延伸(13)。首先我们认为Byte、Word等基础数据类型是受位宽限制的顺序地址存储，当我们把位宽限制这一条件去掉——或通过长度值来指定连续区域的大小——之后，就得到了数组的概念。由于它自然延伸了（但未改变）顺序地址存储的概念，因此它也可以作用于上述这些基础类型。例如：

□ DWORD，等义于长度为4的字节数组；

□ BYTE，等义于长度为8的位数组；

□ INT64，等义于长度为 64 的位数组，或长度为 8 的字节数组，或……

基于此，我们也可以用数组这一概念来统一所有的基础类型，这最终可以将任何数据理解为位数组（bit array）。当然，需要强调，这里的数组是指一个连续空间中的数组，否则就与我们此前的抽象不一致了。

然而，我们留意上述的“某种相同数据”这一抽象限制条件，也就意味着，我们必然会面临“连续空间中包含某几种不同数据”的需求。我们做出这一“必然”判断的原因是，我们的需求总是问题的全集，而非某个部分（所谓可能出现的，必将出现）。因此对问题的某一个分类中的所有可能性施以数据抽象，则它必然可以表达问题的全集，以及满足其背后的全部需求。推论上述逻辑：

□ 设定：在连续空间（S）中，要么包含同一种数据（m），要么包含不同种数据（n）；

□ 如果存有混杂，则可以将它分解为多个连续的连续空间（Si），使 Si 符合上述设定；

□ 如上，我们总是可以用 m 与 n 来表示连续空间中的所有数据。

结构体（struct type）指的就是包括某几种不同数据的连续空间(14)。这一概念是对数组的补充，他们一起构成了“用基础数据类型”来复合其他类型的全部可能性。

五

上述的“全部可能性”事实上还应当包括一种泛义的数组，亦即是元素的类型为某种结构体的数组。在数据结构上，它通常被称为顺序表（sequential list，或 list）。

设数组 A 的每一个元素的数据类型为 T，基于此前的讨论，元素（结构体）A[n]必然有一个确定的长度值 Size(T)。由此，数组的长度——顺序表中的记录数 RecCount——决定了整个数据所占用的连续空间的大小：

在该连续空间中，可以通过数组下标——顺序表中的记录号 RecNo——来访问任意元素，它的地址也是确定的：

其中 RecNo 的取值空间为一个序列值[0…RecCount-1]。

顺序表具有边界判断简单、能快速存取指定位置的特点，到目前为止仍然是关系型数据库的最基本的、最佳的实现方案。关系型数据库中的表格（table）与顺序表在抽象含义上是相同的：每行——每笔记录——的字段列即是数组元素的结构类型定义，行号（RowId）即数组元素的索引下标。因些，结构化查询语言（SQL）中的一行代码：

与将 A 作为数组来存取的时候所采用的操作：

A[5]

是完全等义的。

六

综上所述，我们事实上可以用两种方法来统一顺序存储，由此将一个无限大的空间视作空间连续的数据，并使用地址来存取其中任何数据分量。这两种方法是：

□ 将包括某种相同数据的连续空间视为数组 A；

□ 将包括某几种不同数据的连续空间视为结构 S。

由于 S 本身是连续的，所以：

□ S 可以视为有且仅有单个元素的 A。

所以：

□ 因此整个空间可以统一为 A。

在得到这样一种数组概念的同时，我们也找到了在与计算系统交互时，表示数据的基本方式。它引申数列概念以通过下标索引值来定位数据分量，因此也被称为索引数组（index array）。

在不讨论存储的、纯粹抽象的数据结构的概念集中，我们将索引数组的概念加上结构体，就看到了顺序表（list）；加上操作方式，就看到了栈（stack，LIFO）与队列（queue，FIFO）。

七

顺序存储中用到了两种数据结构概念，即数组和结构体。在此前的讨论中，我们预设了两个前提，其一是顺序存储，其二是数组元素的长度总是能预知的。后面这个条件往往并不能满足，例如我们有一个程序的功能就是“让用户手工输入元素长度”，那么在编写该程序的时候，（程序员）就无法知道如何分配存储了。

这种情况下，程序员在编程的时候知道一个标识，并且要基于该标识来编写后续逻辑，他只是无法将标记与可能的值关联起来而已。例如下面的形式代码：

顺序存储的限制条件需要“地址+长度”两个条件，而上面的例子意味着在程序正式运行起来之前只有“地址”是可以预设的。因此我们只能将该地址“预占”起来（记得排排座故事中的“冬冬不在留一个”吧），以便将来用于找到真实地址。下面对比这种情况与一个实际的顺序存储之间的差异。首先是以数组为典型的存储效果，由于长度是已知的，所以程序员可以安排自 X 位置之后存储其他数据（见图 2-7）：

图 2-7　已知数据内容的顺序存储

然而，在待处理的数据长度未知时，由于“用于存储一个地址值”所需的长度是可知的，所以我们采用“预占”存储效果如图 2-8 所示：

图 2-8　数据内容未知时，预占“存储地址值”所需的确定空间

此后程序员仍然可以安排自 X 位置之后的存储——当然，也可以通过编写程序来设计其后的安排。例如，在程序运行过程中，我们在预占位置填写一个实际的地址值，以说明实际的数据的存储位置，如图 2-9 所示：

图 2-9　保证顺序存储的方案：通过填写预占位置，指示实际的存储位置

通过图 2-9，我们也可以发现，pArea 本身和 Area 本身都是连续的。如果 unknown 部分也按照上述规则来处理，则 unknown 也将是连续的。如此，我们仍然可以保证整个存储空间是连续的。

现在，我们知道这里所谓的 pArea 就是指针（pointer），它是对顺序的结构化存储这一方案在运行期的一种补充，满足两个条件：

□　它是一个标识，有一个计算系统访问它的地址（记为 p）；

□　它包含数据（值），该值是一个（与它关联的、实际的）数据的地址（记为 p^）。

在程序中，通过上述方式来设计数据结构时存在两种可能。其一，如果一个地址是可以先期知道的，那么 p^与值的绑定是静态的、在系统运行之前发生的，因此它也是一种安全的指针。其二，如果一个地址是不可预知的，那么 p^就需要在运行期再动态地绑定它的值；在正式绑定值之前，p^就是一个游离的、无值的标识。

如同我们此前所说过的，如果标识与其值未能绑定，则它的抽象含义——或计算中的数/数据的含义——就是不确定的，这也意味着，在这样的模式上建立的计算系统是不安全的。换而言之，指针的动态绑定（以及解除绑定）是一种不安全的机制。

第二节　数据结构：散列存储

一

排排座的主意其实并不太好：我们总是要尽可能将小朋友排到最前面，否则空缺一旦多了起来，老师们就不大可能记得住了。

于是老师们想起了一个故事。据说，在很久很久以前，有个楚国人在坐船渡河的时候，身上所佩的宝剑掉到水里去了，于是他在船上刻下了一个记号，说：这是我掉剑的地方。当船停在岸边的时候，这个楚国人就跳下船去，沿着这个记号往下找啊找啊……

不对，好像故事讲错了——这个可怜的楚国人好像找不到他的剑了吧？

老师们很快意识到这个问题，于是立即换了另一个著名的故事。又据说，在很久很久以前，有人送给曹操一头大象，但当曹操问这头象有多重时，却难坏了他的官员们——仍然是据说，提出把大象砍成许多段、排成排、逐一称量的那个家伙已经被推出去先砍成许多段了。这时呢，有个叫曹冲的小朋友跳出来说，我们先把大象推到船上去，根据水面位置在船上刻个记号；然后再把大象换成许多石头，直到船沉到记号的位置；最后我们称一下那些石头不就行了吗？

咦！问题解决了！

但新的问题又出现了：为什么同样是做记号，楚人的法子行不通，曹冲的法子却行得通呢？更进一步的问题是：哪种情况下，做记号的法子才确保能行得通呢？

二

所有的数据最终都可以被抽象为数组，或者说我们此前讨论的所有数据类型都可以视为数组类型的转义。我们也讨论到如果一个数组能确定长度，则它总能在运行前被顺序存储，如果它不能确知长度，也可以通过指针来保证在运行中实现顺序存储。

但是我们似乎多了一个题设。我们需要设问：顺序存储是必须的吗？

顺序存储的必要性在于：

□　运算器需要通过一种简单的方法来找到数据；

□　可以通过地址的余量来确知存储的占用情况，进而给程序员控制它的可能。

但是这两项条件都是与机器实现有关的，前者是运算器寻址的需求，后者是存储器容量的限制。而这些——我们必须强调在思想上要突破所有的限制——不是“计算”所必须的。换言之，一个计算过程仅仅是需要数据，并没有限制用何种手段来获得数据。

据此，只要其抽象结果不会影响计算过程，我们就可以将数据的那些仅仅应用于具体语言实现的抽象概念一一抽离。仍以下面的代码为例：

这一句代码（的语法），对应着四个语义：

□ 有一个标识，记为 aNum；

□ 其数据类型为 Number；

□ 标识所指代的数据，其值是可变的（即，变量 var）；

□ 该数据当前值为 100。

当我们把“值可变”与“当前值”这两个——由存储问题推导出来的——概念抽离之后，一个数据就只有三个普遍含义了：

□ 名字，即标识：aNum；

□ 值，即数据：100；

□ 类型，即数据类型：Number。

此前我们也提到过，顺序存储中我们可以将所有数据类型都视为数组的转义，因此我们也可以把“数据类型”从这样的系统抽象中抽离——当一个事物不存在类别含义时，也就无需识别它了。这样，这个数据抽象系统就只剩下了(15)：

现在，我们来到了“名/值”（name/value pair）数据的世界。

三

使用“名/值”关系来确立的数据抽象系统，既有可能确切地找到数据，例如曹冲称象；也有可能找不到数据，例如刻舟求剑。但是这一抽象准确地反映了计算系统的真相：找到数据，计算。

本质上来说，索引数组也是这一抽象下的产物，只不过我们是使用地址来找到数据罢了——

地址，也可以视为数值体系下的标识或命名。而我们现在，只不过要假设命名的方式不再是地址，而是一个真实的、可以被自然语言理解的“名字”，例如字符串。

第一个问题，地址是有限大小的，它与我们的计算系统的存储一一对应，因此使用地址来指示一个数据时，不可能因为越出存储边界而找不到数据。但是我们使用字符串来命名数据的时候，就不能确保数据与存储的这种关系：我们有无穷的方式来生成名字，也就意味着它对应一个无穷尽的数据集合。而这是无法被一个物理的、现实的计算系统——例如计算机——所支持的。

可以用数学方法将一个组合方式无穷的名称集合映射到一个有限的空间中去。这种方法称为哈希（hash）。简单地说，它使用一个函数来为每一个名字生成一个有限空间中的索引，例如数字值(16)。这涉及两个问题，其一是映射为数字的索引值；其二是限制在有限空间中。对这两个问题的一种简单的处理，是将字符的编码值求和、取余。例如：

对于某笔订单的基本信息，其字段名的实际运算结果如下：

图 2-10 表示上述关系在存储上的映射：

图 2-10　一个简单的哈希算法在存储上的映射

它表明，我们可以通过 hash_sum_16()找到(17)：

□　名为'OrderNum'的数据的值为 8；

□　名为'OrderPrice'的数据的值为 1202。

由于求模取余之后，hash_sum_16()的可能值为[0…15]，所以我们能预先分配上述整个索引表区间（16 个基础类型的数据，或者结构体等）。更进一步，我们也可以通过指针来弥补索引表对未定长度数据支持不足的缺憾——这是因为该表本身是一个索引数组，如图 2-11 所示。

图 2-11　指针在哈希表上的运用

根据此前的讨论，我们可以保证图 2-11 中的索引表、数据 1 与数据 2 以及整个空间仍然是顺序存储的，进而不会与计算机的物理实现冲突。

在完整的模型描述中，我们称 hash_sum_16()这类函数为哈希算法或哈希函数（hash function），将名字称为键或哈希键（key，hash key），将索引表称为哈希表或桶（hash bucket），将表中的元素 C0…Cn 称为哈希码（hash code）。图 2-12 描述了这些抽象关系。

图 2-12　哈希表的结构与概念间的抽象关系

四

显然，（图 2-12 右侧抽象概念中的）“键/值”作为抽象系统，的确可以让我们通过标识找到值。我们甚至可以为索引数组在这个系统上的抽象定义一个函数——换言之，索引数组也就是这个系统的一个特例：

但是我们也发现正因为索引数组是自映射的，所以它的映射关系是一对一的（传入 X，返回 X），而此前的 hash_sum_16()却是多对一的，例如'OrderNum'映射为 12，但如果传入标识'Order'，其映射的结果也会是 12。

我们面临要在同一个位置上放两个或者更多数据的情况。这就是哈希冲突（哈希碰撞，hash collision）。对此我们可以增大哈希表的长度，但即使它超过可能的哈希键个数，（在不同的算法下）仍然存在冲突的可能。既然我们承认冲突必然存在，因此在找到 Value 之后再做一次验证就是必需的了。这里的验证也有许多种做法，其中之一是用新的一种算法再做一次 Hash()。尽管两个不同的 Key 在两种不同的 Hash()之后仍然相同的机率相当低，但是——仍如此前所述的——还是必然会存在。所以最终的一个步骤，通常还是对 Key 的直接识别——例如字符串的逐字符比较，或者数据存储区块的逐字节比较，或者基于顺序存储的、区块地址的比较。这种情况下，数据区必须保留原始的 Name/Key 值，如图 2-13 所示。

图 2-13　验证：保留 Name/Key 的原因

有两种特殊情况，其一是元素在系统中不一定完全出现，但其可能值是预知的，例如一周的七天；其二是元素总是会完全出现在系统中，例如键盘上的所有键。这两种情况所面临的数据集都是有限的，对此我们仍有机会设计一个函数来使得每一个 Key 所计算的 Code 都保持唯一。这样的哈希表是静态大小的，其长度为数据集范围的上限，即：

这其实是为每个元素创建了唯一的哈希码映射，只需要比对哈希码值即可确认数据，也因此就无须再保留原始的 Name/Key 值了。这同时也节省了图 2-13 中最后一步比对 Name/Key 的开销。

五

曹冲称象本质上是通过一个系统将大象的重量映射为石头的重量，而哈希算法正是这样一个系统。对于象与石头的重量来说，这个映射关系是确定的、一对一的，因此刻度也就只需要简单地核对，而无须“原始的象”来参与复核。

但在另一个故事中，楚人为什么就失败了呢？

因为哈希算法背景变了。需要留意的是，楚人的哈希算法：

□ 求掉落位置之于船体的相对位置

没变。算法的结果 HashCode：

□ 刻度

在整个过程中也没变，但是楚人计算刻度时的背景是船体，使用刻度时（下水找剑）的背景却是整条河。所以“名/值”数据系统与其存储背景有着相关大的关系，“必须在相同背景创建与维护哈希表”是一种系统负担，这种负担通常是由语言或某种硬件装置(18)来维护。

对于在某种特定背景下建立的一种“名/值”关系型的数据系统，我们称为关联数组（associative array）。这种抽象的数据结构在基于地址存取的机器中应用时，面临的核心问题是：

作为名字的字符串存在无限的组合，这与有限的地址空间是矛盾的。

而哈希机制/算法通过将“名/值”映射为“Key/Code/Value”的关系，从而解决了上述问题。这一映射关系是“哈希算法+存储背景”来约束的，这既是系统负担，但也使系统从根本上避免了刻舟求剑的笑话。

解决了与地址空间的矛盾之后，关联数组得以在顺序机器中实现。它在物理上仍然能够满足顺序存储的需要，但从逻辑上却分离了名字（标识）存储与值存储的关系。而我们知道，标识与值是数据最重要的两个性质。

总的来说，如今我们在计算世界里使用的数据表示方法只有两种，即关联数组和索引数组。无论是在抽象概念还是具体实现上，索引数组都可以视为关联数组的特例，既有存储限制，也有存储限制带来的名称/标识限制。

第三节　执行体及其执行过程中的环境

一

河岸。路人甲。

如何过河是一个问题。一般来讲，其他可能的选择是不过河或者绕过去，但前者不是甲当前的选择，后者则充满了变数。对于聪明的路人甲来说，他找到一棵树，挖空了树干，做成了一艘（原始的）船。于是他到达了对岸，离去，留下一个名为“船”的东西(19)。

如何行船以到达对岸？这个问题的解被路人甲带走了，尽管船还留在那里。如果你能从路人甲——或者其他人、使用手册或者你努力地思考——那里得到这个问题的答案，那么你就会用船过河了。

总有些知识是可以复制的(20)，例如船和行船的方法。复制这些知识，就可以得到一个算法；只要条件合适，就可以得到相同的解。

那么，什么才是合适的条件呢？

二

下面是“一艘船”：

这艘船造得并不怎么好，但这是一个稍晚一些才会讨论的问题。在这里，我们先关注其中的要素，这包括既存的 ship、people 与 water。其中，ship()封装了：

□ 知识 1：row 是 people 的一个行为；

□ 知识 2：row 这一行为的使用；

□ 知识 3：checkShore 这一检测行为的使用；

□ 知识 4：不断 row 直到 checkShore 得到正确结果这一过程。

除了 ship()自身封装的上述知识之外，以下知识也是存在的：

□ 知识 5：如何实施 row 这一行为；

□ 知识 6：如何实施 checkShore 这一检测行为。

更进一步，上面的 ship()还隐含了一个不确切的知识：

□ 知识 7：checkShore 是谁的行为？

如果我们（我的意思是说，作为 people 的行船者）正确地理解与支持上述知识，那么 ship() 可以帮助你到达对岸。

我想已经有程序员开心得大叫起来：总算看到了“面向对象编程”（OOP，Object-oriented programming）了。

没那么快。

三

因为我们得先讨论行为。如前所述：

□ ship 本身（包括知识 2、3、4）和知识 5、6，是定义行为；

□ 知识 1、7 是找到行为；

□ ship()、row()、checkShore()是行为(21)。

行为总是未可知且无穷尽的，例如路人甲面对河岸时的可能选择与解。但是“定义”与“找到”行为的方法则可以确定。对于前者（定义行为），它与声明一个变量本质上没有区别，例如：

对于后者（找到行为），如果 ship 可以像变量一样被定义，那么我们找到它的方法也就与（此前我们讨论过的）“通过标识找到值”没有什么不同。换言之，

总是可以将变量和方法统一为数据。

四

回顾上一节的所有讨论，我们在语言中使用一个数据的方法，根底上只是如下过程：找到它，使之参与运算。而关联数组使“找到数据”这件事变成对一个计算背景的维护。例如，我们有一段代码：

这些数据的定义可以被理解为一个背景的建立（当然，我们也可以为零个数据建立一个背景），因此我们得到一个关联数组：

接下来我们在这个背景环境中运算。但——根据语言的不同——我们可能又需要“即用即声明”一个数据，例如：

而 i 这个数据的出现，意味着我们需要在 aAssociativeArray 中添加一个新的 Name。虽然 i 的值是可变的，而在整个过程中 i 的名字却不变，因此我们对于 aAssociativeArray 的 Name 只有添加和删除的需求，不需要因为值的改变而导致 Name 的改变。更进一步，

我们事实上是将一个数据的生存周期映射成了一个 Name 的增删。

五

接下来我们讨论与此相关的四个问题。在讨论语言与关联数组的性质的过程中，这些问题分别在以下章节中出现过：

□ 第五章 语言及其面临的系统，第一节 语言（五）

□ 第六章 程序设计的核心思想，第二节 数据结构：散列存储

其一是增删 Name 的必要性，即如果一个语言认为变量是计算的前设（不支持变量的动态声明），则我们只需要一个静态的关联数组来维护其背景，否则我们需要一个动态维护的关联数组。这意味着我们在静态语言和动态语言(22)中都可以使用关联数组。尤其重要的是，对于静态语言来说，我们可以在编译期使用关联数组，而无需在数据区保存一个用来复查的 Name 列表。

其二是 Name 作为前设，但其值仅在计算过程中可知，这一绑定关系可以变成在上述背景中的一个值的修改。也就是说，

可以映射为：

与此相关的，a 的确定性（是否可以改变值）也可以映射为对 aAssociativeArray 中的名字 a 的访问限制，即 aAssociativeArray['a']是否可写。

其三是如果我们需要在计算过程中新建一个数据，它的含义无非是我们要在上述关联数组中添加一个“名/值”。例如我们在上述背景中执行如下代码：

如果这行代码得以执行，它表明需要动态创建 x 这个名字，我们只需要将该 Name 添加到

aAssociativeArray 中，并置值为 1000 即可。当然，这也相应要求 aAssociativeArray 是可以动态维护的。

其四，如果我们试图在运算过程中访问一个并不存在的名字，对于没有这一项需求的静态语言来说，aAssociativeArray（可选的）可以不保存 Name 列表；对于动态语言来说，它将表现为在 aAssociativeArray 中无法查找到某个 Name。

可见，对于一个计算过程来说，关联数组可以维护它所需的一切参考，所有的数据性质都可以表述为关联数组与其存取的性质。我们称这一运算所需的参考为上下文环境（context，context environment）。上下文是保证一个运算具有确定性的主要方式，换言之，它相当于语言中的语用这一要素。对于计算来说，在此前的讨论中我们说过“数据确定，则运算确定”，因此关联数组本质上只是为计算维护了一些数据而已。

六

对于 ship()来说，所谓合适的条件就是类似这样的一个上下文环境：

我们来考虑类似环境的可能性。例如，假设我们将原始问题中的“水”和“检测岸”确定下来，则 ship 将可能有许多种方案。如：

假设我们也将“人”确定下来，那么方案就只剩下一种：

除非我们——人——能有不同样的 row()方法。

在从_ship_env_0 到_ship_env_1 的演变过程中，我们将 ship()与环境绑定在了一起。这是一个不小的变化，因为它的含义是语义与语用之间的绑定关系。如果我们将_ship_env 分离出去，则 ship()将是 DSL（领域特定语言）的一个实现；如果将_ship_env 与 ship()绑在一起，那么 ship()就是一个应用程序中的确定求解。

前者创建了一门语言，后者创建了一个程序。

七

我们已经讨论四点：

□ 关联数组的实质是“名/值”映射；

□ 代码中的数据与逻辑可以统一为数据，进而统一为标识；

□ 关联数组可以维护一个计算过程所需的参考，亦即上下文环境；

□ 对于静态语言来说，上下文环境的实质是一个（可选）不需要名字/标识的关联数组——这种情况下，它更像一个索引数组。

综上所述，对于静态的、编译型语言来说，下面的代码：

意味着 P()这个计算过程所需的环境有两个，其一是 x1，x2 … xn，它们在 P()进行计算之前就可以预知；其二是 y1…yn，它们在 P()计算的当时才会得知。对于编译型语言来说，它认为：

□ X 是 P()计算的前设；且，

□ X 与 P 本身是后续其他计算的前设；继续推论之，则，

□ 所有在最外层出现的标识是整个系统运算的前设。

由此，编译型语言通过编译器程序

□ 将所有这些标识与其内容——例如数据、函数（逻辑、行为），

□ 根据所有计算过程约定的计算环境——例如操作系统，

□ 按照确认的规则——例如可执行文件格式，

编排并放在一个实体文件中。最后，操作系统（桌面用户或服务进程）决定何时将该实体文件装载并运行（Launch）起来，以完成所需的全部计算。

我们看到上述过程的限制条件包括：操作系统环境、可执行文件格式和编排方法。这些是程序得以执行的要件，但并非我们需要讨论的内容主体。因此，图 2-14 以 32 位 Windows 为例，仅大体说明这些要件之间的关系：

图 2-14 从一个入口开始：操作系统准备的执行环境

参考图 2-14，对于操作系统来说，仅需要知道一个程序的“入口代码位置”。例如：

随后，入口代码将按照规则在“代码表”中查找标识，例如：

最后，如果 P 或 main 需要数据，则使用类似的方法通过“数据表”来查找标识。

编译型语言在程序执行前就明确这些标识，因此编译过程可以省去这些名字，而指代以存储中的索引，亦即地址；因为上述规则是操作系统对存储的约定，所以对于编译型语言来说，该可执行文件（PE 文件，Portable Execute）可以置入存储环境中，以应用上述地址；因为操作系统知道上述 PE 文件的入口地址，所以只需要：

□ 装载该 PE 文件、

□ 分配地址、

□ 交出 CPU 的执行权限，

然后就可以完成整个计算过程。

可见，节表是节的索引，是节中的数据与代码(23)（及其标识），是整个计算过程的参考——上下文环境。推而广之，对于整个进程来说，它可以通过导入表来获得整个操作系统的上下文环境；细究之，对于一个内部函数来说，它可以通过入口参数表来得到上下文环境（例如此前讨论的 y1…yn），进一步也可以得到进程的、操作系统的上下文环境。

所有的行为仅仅是(24)：1、找到（包括逻辑在内的）数据；2、计算。

那么对于解释型语言来说，上述过程是更复杂，亦或更简单呢？

第四节　语法树及其执行过程

一

我们已经在讨论一种具体的语言实现形式，即所谓“静态的、编译型”语言。这事实上涉及到两种语言的分类法，其一是静态与动态的，其二是编译与解释的。

在计算机语言实现中的所谓“编译”，是不同于翻译（translation）的。从本质来说，我们所

使用的任何计算机语言编写的代码都需要通过翻译才能交由计算机执行，因为计算机最终是只能理解开关状态的电子电路。但是在最终抵达目标语言——即机器语言，或称之为电子电路的序列行为——之前，我们事实上会经过不止一次的翻译过程。在这些翻译过程中，编译（compile）与解释（interpret），都不过是其中的“某一个过程，或包括多个过程的阶段”的代名词而已。

问题是这个翻译过程可能有相当多的步骤，而且今后步骤还将有增减（或因系统复杂而变多，或因优化而变少），所以几年前的编译或解释（这些过程）与今天就可能根本不同。为了避免这类问题，我们采用两种极端的方法来定义它们，即对于一门语言翻译结果的执行体：

□ 如果是可以直接指示 CPU 行为的指令，则我们称之为编译型语言；否则（必然地），

□ 它需要被某个软件再经过（至少一次）翻译才能得出上述指令的，我们称之为解释型语言。

“静态与动态”这样的分类法，源起于语法与语义之间的绑定关系。其中所谓的语义包括数、数据与逻辑等。一般来说，语言中会通过“有值但没有标识”的数据来表达“数”这一概念，即(25)：

□ 对于没有标识的代码（这种数据），我们称为匿名函数（或匿名方法等）；一般性的没有标识的数据，我们称为直接量（或字面量、立即值）。

接下来，我们前面所谈到的数据结构最终被表示为（语言中的）数据类型，数据类型也因此可以理解为数据在语言中的性质——除非一个数据是无结构的。那么将语义绑定于标识时(26)，事实上语法元素就有了对这些值、数据类型以及对逻辑（代码、执行体）这一特殊数据的考量。作为一个约定，如果在代码执行之前

□ 代码与该段代码的标识是绑定的，并且

□ 任何上述代码所使用的数据的标识具有确定的数据类型含义(27)，

则我们称该语言是一个静态语言。

二

相反地，一个语言环境中也可能存在四种与上述约定不相容的情况。

其一，在执行之前存在标识但没有任何已知的值，代码运行时该标识存有绑定任何值的可能，称为动态绑定(28)。例如下面的代码行在 JavaScript 中表示环境中有一个标识 age，但直到该行代码执行时，age 所指示的存储中才会有值 20：

与此相对，我们称标识在运行之前就具有了值和类型的情况为静态绑定。例如在

Delphi/Pascal 中：

其二，对于已经（动态或静态）绑定过的标识，如果通过写值的方式能够使之具有新的数据类型含义，我们称为动态重写(29)。对此前的示例 1，我们可以有如下代码：

这会导致 age 具有新的数据类型，而这在 Delphi/Pascal 中会被视为违例。

其三，动态绑定或是动态重写导致标识具有不同数据类型含义的情况，称为动态类型。示例 1 中的 age 在动态绑定发生之后，JavaScript 环境允许使用 typeof()取得其类型为 Number，但在绑定发生前则是未定义的(30)。

其四，如果允许将（任意或特定数据类型的）数据作为代码片段加以执行，则称为动态执行。例如：

当然，这也意味着多数脚本语言具有动态执行的性质，因为它能读取文本文件内容并执行。

对于支持上述（“静态绑定”是显然被排除在外的其他四种性质）四种性质中的一种或多种性质的语言，我们就称为动态语言。

所以我们此前将部分讨论限定在“静态的、编译型”语言之中，实在是有着严苛的条件：既要能直接编译成机器语言，又要能明确有标识和数据类型含义。接下来，我们的讨论则要广泛得多。不过此前讨论的那些性质，在后续讨论中也仍然是合用的——这本来就是我们构划这样一个讨论路径的原因。

三

编译过程将代码变为机器指令，这使得代码必然面临机器环境的限制，例如存储的使用与 CPU 执行权限的交接就是这类语言中主要的两类复杂问题：内存分配和线程调度。解释型语言将程序的入口由 CPU 指令序列变成代码的自然入口，散列存储使我们避免面临存储地址的限制，这些技术可以让语言从“机器问题”中解放出来。这一结果让我们认识到，机器环境的限制只是语言实现中的负担。

因此，我们的问题得以回到此前的讨论：程序的所有行为无外乎两点，其一找到（包括逻辑在内的）数据；其二计算。亦即是说，在一个解释性的，或不以机器问题为焦点的语言中，“找到（包括逻辑在内的）数据”必然会成为本质、核心和关键的问题。

表 2-6 列出了整个系统中的语法元素的性质。

表 2-6　语法元素的性质

* 指定行为所依赖的行为定义。例如 JavaScript 中的排序，可以指定排序过程。

** 作为通用行为，必须传入行为具体过程与参考数据。例如 Win32 SDK 中的 EnumWindows() 的列举。

*** 返回同时包括数据与行为的一个结构。例如 JavaScript 中的闭包。

通过“定义”，我们确定系统中的元素必然只存在数据、行为。这种定义通常是语法性质的，但最终总是可以表达为一个“名/值”映射——虽然这并非实现上的必需。需要留意的是，尽管表 2-6 中刻意从“数据”中把“行为”分离出来，并将后者进一步地分解为：

计算()→值

这样两种，而事实上它们都可以视作数据，用相同的方法定义，并施以相同的行为。

所以我常说，程序是“被定义出来”的。因而我们可以通过思维完成全部程序，而它在执行过程中的排错、修正与优化等，是出于我们在工具使用上的、熟练度训练的必需，而非编程思维训练的必需。

必须在这里指出的是，无论我们的编程语言如何变化，上述都是语义完整性的需求，即最大可能的集合。当然，其前提在于我们对“定义”的假设，如果所定义的语法元素更多，那么这一组合的空间就会变得更大，在语言的语义表达上会趋于丰富，进而可能无法控制；如果所定义的语法元素变小，例如将数据与行为统一为数据，则语义表达上会更简洁，但也可能出现（用自然语言来理解时的）歧义(31)。

最后，我们也可以推论出：既然“找到（逻辑与数据）”本质上也是一个行为——或称之为计算，那么它必然也满足上述的 2～3 的全部约束，亦即它可能实现的全部方法(32)。

四

所谓翻译（编译与解释）无非是将一系列的源代码文本（所对应的计算行为）排列成一个顺序序列，而最终的执行器——无论是机器还是解释器——只是从这个顺序序列的起始位置开始处理，一直到结束(33)。正是因为这个缘故，所以依据什么排序才会是翻译中的一个关键问题。

语法树是对语法元素进行排序的一个方案。这一方案将语法元素解析为不同类型的树结点，

例如操作符结点、操作数结点等。以下面的代码为例：

它可能被解析成如图 2-15 所示的语法树(34)。

图 2-15　语法树：对语法元素进行的排序

在该语法树中，所有叶子结点都是标识或直接的值——它们代表数据，而非叶子结点则是操作，亦即逻辑；如果是逻辑，则计算的结果可以作为其他计算所需的数据，例如图中的结点 8 的计算结果是结点 4 操作数。

当如上的语法树形成之后，顺序扫描树中所有结点的方法有两个：深度优先遍历与广度优先遍历——理论上说存在无数种可能的方法，这取决于形成该树时所使用的算法。以使用深度优先遍历算法为例，我们将得到下面这样一个处理顺序：

去除掉标识和数据部分，我们看到的是：

这几个操作的顺序，即我们对“顺序计算（逻辑）”在语法树上的重现。

那么，以数据排序又会是怎样的一种情形呢？例如，下面的代码：

可以产生一个类似的语法树，如图 2-16 所示。

图 2-16　生成的语法树（为分析“基于数据的排序”而进行了排版变化）

（除了版式上的不同）这个图与此前的图并没有本质的区别，我们仍然可以按上述遍历规则得到一个执行顺序——如果你将这个图理解为基于运算的排序的话。但是，我们这里要讨论的是基于数据的排序，即：运算影响到哪些数据，以及其影响先后的问题。基于此前的讨论，我们可以将“数据”理解为运算所需的参考——上下文环境，如图 2-17 所示。(35)

图 2-17　基于数据的排序：将“数据”理解为运算所需的参考

□ 将图 2-6 中树的每一个运算放到它所需的上下文环境中，并

□ 将这些环境用框图及其层次表达出来(36)。

可见，本质上来说，我们要正确处理此前的代码文本，则意味着我们将对上下文环境的相关性进行排序：内层是被依赖的，因此排序优先级更高；同层不存在数据/上下文环境依赖，因此优先级相等（例如 1 与 3、2 与 4）。

比较两种方案：以数据为顺序时，标识符的作用域（亦即是上下文环境）问题将会绑定在语法树上；而以逻辑为顺序时，作用域问题与语法树是无关的。

第五节　对象系统：表达、使用与模式

一

所有被计算的数据，我们要么看做是顺序存储的值，要么看成是散列存储的“名/值”。但这只是对数据全体的一个认识。针对其中一个局部，例如某几个地址上的连续数据，或某几个“名/值”数据，我们又如何认识它呢？它们在数据概念上又是什么？

在顺序存储中，我们已经为这样的数据找到了一个“看起来合理”的名字：结构类型/结构体。基于这一存储方式的特点，我们对结构体中的“字段名”并不敏感，例如说：

这个结构有 first 和 age 两个字段名，但忽视这两个字段名，我们以下面的结构体来操作它：

也没有什么不同。更进一步，它与一个字节数组也没有什么不同(37)：

但是散列存储中的某几个“名/值”数据就不可能这样消化掉了。例如：

其中 name 与 age 这两个名值对，其内部是有依存关系的——它们分别是 info 的不同性质；也有其外延，即可以作为其他的数据的性质，例如：

如我们此前讨论的，name、age（以及更多个）这样的名/值构成了一个数据系列，并满足了我们对数据的内涵与外延的定义，因此它必然可以被理解为数据。而其作为数据，其何种系列的抽象，亦必将为一种数据结构。

这种系列的抽象，我们称为对象。对象是一种数据结构，其每一个分量数据为一个属性，属性可以是对象、索引数组、关联数组，或它们基础的、延伸的数据类型之任一。

对象可以看成是关联数组的一个自然延伸。我们知道，关联数组在概念上与“去除存储限制的”索引数组可作类比；与此相似，对象与“去掉存储限制的”结构体也可作类比，图 2-18 表明了这样的关系。

图 2-18　“结构”与“对象”在抽象本质上的一致性

可见，数据的最终抽象，究竟是“结构体”还是“对象”，只是缘于我们在“如何找到数据”这个问题上的不同选择(38)而已。除开这一点，二者殊途同归，在其抽象本质上是一样的(39)。

二

对象是带有一系列相关性质的数据。除此之外，它不带有任何必须附加的概念。明确这一点，有利于我们看到对象系统的本质与构建过程。

在《结构程序设计》中，达尔(40)用一种非常奇特的方式来定义了类与对象之间的关系，即：如果一个过程能够产生比调用语句存活得更久的数据实体，则我们称该过程为类；这样的数据实体（实例，instance），我们称之为对象。更有趣的是，《结构程序设计》将这样构建对象系统的过程称为“层次程序结构(41)”。不过这其中的“层次”却并不是指类的继承层次，而仅仅是指“按层次的方式构造和分析”去处理系统的复杂性问题。达尔在书中强调了“层次方式”是唯一有效的办法，但并没有认为系统的逐层分解与对象的继承之间存有某种必然联系。

事实上在 Simula 67 最初的“面向对象”观念中，对象只是一种（相对于一般数据类型而言）更为高级的数据抽象形式。直到 1971 年 SmallTalk 才提出了继承性概念。如今，对象的 PME（Property-Method-Event，即属性、方法、事件）模型，以及 EIP（Encapsulation-Inheritance-polymiorphism，即封装、继承、多态）构成了完整的面向对象编程的概念集。

但这些都并不是对象这一概念抽象的本义，而是实现对象系统过程中的一些实践。例如，我们可以将基于继承的对象系统，视作是通过层次方式来处理系统复杂性的一个实践。准确地说，它通过类属关系从开发目标复杂无序的数据中抽取了一部分出来，使它们成为一个可编程的、（在一定程度上）可复用的对象集。正因为这一过程只处理了具有类属关系的层次数

据，所以——必然地——下面这些问题也就局限了面向对象系统的应用：

□ 无类属关系的数据；或

□ 非层次的数据；或

□ 系统中的逻辑需求。

三

将对象作为“一种数据”来观察，则所谓 PME 中的方法（M）与事件（E）就可以视为是“面向对象逻辑需求的”特殊属性（P）。其中，方法是对象创建之前即可确知的自有逻辑，事件则是在对象创建时仍有未定因素的外部逻辑，这二者本质上也是对象的性质。如图 2-19 所示。

图 2-19 PME 作为语言特性所实现的抽象概念

所以 PME 其实只是概括了对象属性，而无助于得到这一对象本身。也就是说，从 PME 的角度上来思考，只是对对象的细节刻画；如何从目标系统中识别到该对象，则是刻画它的前提。

对象的继承性则是对“对象如何获得”的一个解释。如同我们说整数作为数据，在它的系列中的关系为“+1”一样，继承性约定了对象——这个数据在它的系列中的“关系”为层次继承，如图 2-20 所示。

图 2-20 基本的对象继承

若对象 C 继承自 B，B 继承自 A，则 C 必继承自 A，即 C 必具有 A 都有的属性性质。如同人位于生物继承关系的最末端，因而人必然有所有生物的所有基础性质。又如图 2-21 所示。

图 2-21 对象继承性的衍化与证明

基于以上推论，若对象 A 具有某唯一性质，例如“A 是对象”，则 X、B、C 都“必为对象”。因此，对象的继承性保证了对象系统中的每一个数据都必然是对象。

但是对象 X 与对象 C 又各自具有不同性质，例如人与马具有所有陆生动物的性质，但人与马又有不同。这一点，又是通过对象的多态性来说明的。多态性意味着对象 X 与对象 C 是在对象 B 的基础上相同的，但在其各自的表达范围中却是不同的（多态）。例如图 2-22 说明对象 X、C 在范围{B}中是相同的，但在范围{X}和{C}中各自不同。由此，我们可以进一步推论图 2-23

中的对象 X1、X2 与对象 C 在范围{B}中也是相同的：

图 2-22　X 与 C 是范围{B}中同类对象的多态

图 2-23　继承性决定了 X1、X2 与 C 在范围{B}中的多态关系

由此可见，多态性是对象的性质在其继承关系上的表现。由于继承关系本身即是附加的性质，因而多态性也是附加的性质。

四

我们再来略为讨论一下可见性的问题。

所谓可见，是数据相对于计算（的需要）而言的——如果数据不需要参与计算，则讨论它的可见与不可见本身就失去了意义。我们这里所说的计算，在对象系统中被称为行为/方法（且方法本身也是对象的性质之一）。这些与计算相关的因素，综合起来有如下几种情况。

其一，以一个对象来说，属性若以“被计算数据（P）”和“计算行为（M）”来区分的话，则 P 本身就有对 M 的可见性问题，如图 2-24 所示。

图 2-24　对象的数据之于行为的可见性问题

M1 是否可以访问 P1？若可以，那么 M2 能否访问 P1？这个问题的有趣之处在于，在当前我们对对象系统的设计中，Mx 是能访问 Px 的，因为它们是属于同一个对象的性质。但若 P1 只需要被 M1 访问，那么 P1 是否应该仅是 M1 的一个私有变量？例如代码：

显然，身高与体重都是 aman 的性质，但身高与增高有关，与减肥却没什么关系。所以是否应该考虑 height 是 heighten()的一个私有的、计算用的数据呢？但如果这样，height 难道又不是 aman 的性质了吗？

更进一步说，我们的现状是：让任一方法 M1 都必将面对所有的属性 Px。这在一定程度上增加了对象自身构造时的复杂性。

其二，以一个对象作为系统中待处理的数据来说，则对象本身（与其所有属性）就有对系统的可见问题，如图 2-25 所示。

图 2-25　对象之于系统的可见性问题

因为外部系统可能需要了解 P2 与 M2，而并不需要了解 P1 与 M1，因此 P1 与 M1 就应当对外部不可见（内部私有，internal-private），P2 与 M2 就应该是对外部可见的（公开，public）。再者，与上面的一个问题相关，在 P2 与 M2 中又将涉及到 P2 是否需要被公开的问题。

其三，若一个对象作为继承层次中的一层，则对象有对其他末端层次的可见问题，如图 2-26 所示。

图 2-26　对象在继承层次上的可见性问题

因为对象 C 的实现可能依赖于一些对象 B 的性质，也可能根本就不依赖某些性质。对于完全不依赖的，就不需要由 B 继承到 C，因此这里的“私有”并不单单表明对象 C 是否可见，而且表明性质 P11/M11 是否要从类继承关系上完全隔离开来，确保对象 C 无法通过继承得到。反之，我们就应该让对象 C 可以继承 P22/M22，以确保其他行为可以访问它们（内部——在继承链上的——保护，internal-protected）。

其四，基于上述问题的进一步设问是：若外部系统中有一个对象是继承自对象 B 而得来的呢？如图 2-27 所示。

图 2-27　跨系统继承（对象复用）时的可见性问题

这种情况下，对于 P22/M22，我们仍然需要让外部系统可以在对象 D 中访问到（外部——在继承链上的——保护，protected）。

回顾上一小节的的讨论：多态性是对象的性质在其继承关系上的表现，而本节内容则指出这一表现的具体方法（即可见性）包括：

□　可见但限于在内部实现的子类的，即 protected internal；

□　可见并允许包括外部系统实现的所有子类的，即 protected。

上述分析也表明：如果要在对象继承中，利用子类相对于父类的多态特性，则相关的性质必然是上述两种可见性的。除此之外，其他的可见性还包括：

□　不可见，即 private；

□　可见但与继承性无关，即 internal 和 public。

五

除了以下三种关系，即：

□　对象之间的关系——继承，

□　对象性质在其继承关系上的关系——多态，

□　对象性质分成属性与方法之后带来的关系——可见性。

之外，对象之间、对象的性质之间、对象的性质与对象之间还存在哪些关系呢？

GoF 模式的提出，在一定程度上是对上述问题的回应。GoF 模式的一种基本分类如图 2-28 所示。

图 2-28　GOF 模式的基本分类

“目的”这一视角下的三种分类所描述的正是对象之间的三种关系：

□　创建型：基本关系，继承性的定义与获得；

□　结构型：将对象理解为数据时，除继承关系之外的其他关系，包括组合、分类等；

□　行为型：观察对象的行为能力时，对象关于行为的可能关系。

我们先讨论结构型与行为型，因为它们事实上与“面向对象”没有什么关系。结构型讨论的是从系统的外部来看，一组对象应表现为何种属性集（属性组合）的问题。

例如 Adapter 这个模式，我们先假设一个系统中有 N 个对象，N1 与 Nx 各个不同，但它们都有一部分相同的性质，这些 N1…Nx 就适合用 Adapter 模式向外表达这种相似性。它们各自实现 IAdapter_N1…IAdapter_Nx 接口，只要 IAdapter_N 接口一致，那么外部系统总是能访问到 N1…Nx 的每一个对象的上述相同性质。

如果该系统中另有 M 个对象，且这些对象原本都是继承自相同的基类，它们因此而具有某些相同的性质，那么这个基类就可以被称为一个 Bridge 类。M1…Mx 是该 IBridge_M 的 x 种实现，这个模式以此掩盖了“因为基类相同而具有相同的性质”这一事实。

通过这一分析可见，Bridge 与 Adapter 模式的最终目的，就是向系统外提交“具有一组相似性质”的对象中的一个实例，而系统外并不需要关注其内部的转换、继承或组合等实现方式。

那么，如果我们将上述对象换成数据，是不是也存有类似的模式呢？答案是肯定的。向系统

外暴露一组数据性质并不是一件复杂的事情。以下述数据（一个数组）为例：

设我们为 data 绑定了一组行为，分别用于存取 a1…a4 属性：

那么我们显然可以向系统外暴露一个对象，称之为 Adapter：

该 Adapter 具有 data 数组所表达的所有性质，但 Adapter 自身既可以不是对象（例如作为结构体），也与继承无关。

参照如上的分析，可以得出表 2-7 和表 2-8。

表 2-7　GOF 模式：对结构型的分析

* 尝试以一个对象为基础，增加或隐藏性质，使之表现为这样的接口或对象类型。

** 不依赖于这些对象的个体，因此可以置换对象中的部分或全部，而不改变其上述表现。

表 2-8　GOF 模式：对行为型的分析

* a 以及 a 的行为是可追加的，并能根据外部环境决定如何行使行为。

图 2-29 更清晰地阐述了这一事实，即所谓模式的结构型与行为型，其实是从数据与逻辑的视角观察一组对象或一个对象的结果；而所谓模式，是对上述观察所见关系的一种抽象描述。

图 2-29　模式是对视察所见关系的一种抽象描述

最后，我们也可以更进一步地观察到：（除了 Bridge 模式之外）“对象”与一个一般含义上的、具有一组性质的数据并没有不同。因为我们在图中所说“所需性质”其实就是“具有一

组性质的‘名/值’列表”，这既可以是结构体，也可以是（非继承关系下的）对象。亦即是说，这些模式是否要基于“面向对象系统”去实现是无关紧要的。

六

接下来我们再回过头来，深入地讨论一下对象系统中的继承性问题。

首先，继承是一种抽象方法，即对事物间延伸关系的一种认识。同时它也是一种手段，使得对象的一个性质能够“遗传”给子类对象，从而使子类对象可以具有该性质，而无需特别地、显式地说明。

这涉及对象、类与子类的定义与关系。《结构程序设计》将“类”定义为“产生对象（实例）”的过程，这的确带有历史痕迹。除开这一点，这一定义意味着两个事实：

□ 类，必须了解对象是什么样的，即类必须知道对象所有性质的定义；

□ 对象，必然是由一个构造过程产生的，这个过程必须知道类，或等同于类本身。

基于这两点，如何来实现继承呢？总结如今在面向对象开发上的实践，所谓继承，有三种实现方式：原型继承、类继承、元类继承。

原型继承是对上述第二项的实践。这种继承方式认为，子类对象对父类对象的相似性可以藉由“抄写（复制）”而得到。这一抄写的过程，即是构造过程。因此，如果一个构造过程 F 产生 b 对象，则 F 只需要知道 b 的父类对象 A 即可，并不需要一个独立的类来获得这一认识。下面的代码反映了这一实现原型继承的基本模式：

类继承则显式地要求一个“类”来记录对象间的继承关系。类可以是

□ A：构造过程（逻辑），或

□ B：记录上述关系的数据，或

□ C：仅仅是在源代码期间可知的一种文本声明（定义，由翻译程序处理）。

最后一种情况 C 是基本形式。即类作为某种面向对象语言的一个特性/机制时，仅有翻译程序需要知道代码中对象之间的继承关系，这一关系可以用声明性的文本记录在代码中，而完全不需要成为可执行机器代码（即翻译程序处理的结果）的一部分——因为，事实上我们说过，机器代码根本不知道何谓“对象”，更无所谓的“继承”关系(42)。

当这种关系需要为程序员所知——例如试图在代码中知道和处理继承关系，那么它就必须被表达为一种数据，即类，亦即是上述的情况 B；更进一步，该种数据也必然因其具有一种系列关系而具有数据类型，即类类型。可见，类类型本身只描述、记录对象的继承关系；而类，

根据我们此前的定义，它可以是构造过程本身，也可以仅仅是构造过程执行时了解继承关系所需的一个参考(43)。

那么，从这里也可以了解到，事实上如果构造过程本身记录了上述继承性关系，又可以向外公布这种继承性关系（以便程序员得知），那么类也可以是一个过程，即上述的情况 A。这种情况下，类类型也可以是该过程的一个引用(44)。

下面的形式代码（基于 JavaScript）反映了后两种实现类继承的基本模式(45)。

现在，我们再一次考察支持类继承的系统，便会意识到一个问题：这样的对象系统中既有对象，也有类。既然对象是通过类来得到的，那么“类”又是从哪里来的呢？也就是说，对于最终执行的程序来说，如何了解一个类自何诞生？

而这也就是元类继承所要解释的问题了。既然

□ 我们可以将所有数据视为对象，又，

□ 上述的类也是一种数据，那么，

□ 上述的类自然也就是一种对象，因此，

□ 也就可以有类的构造过程。

这种构造过程，就被称为元类，它是在从对象系统——这一数据定义——中分别出类之后，所必需的一种解释。而本章中有关类的一切解释也可以同样作用于元类，例如元类也可以有上述的三种记录对象间的继承关系的方式。

（在上一小节所述之外的）其他六种 GoF 模式讨论了对上述“构造过程”的实现(46)(47)，但并没有强调继承关系通过何种方式来维护——事实上 GoF 隐含地基于并依赖“类继承”模型。这些实现方式的关系如图 2-30 所示。

图 2-30　GOF 隐含地基于并依赖“类继承”模型

七

在上述对对象系统的讨论中，我们首先考察的是剥离掉继承性之后的对象——亦即一组性质，或称之为一个复合的数据（或结构体）。对于这样的数据，GoF 模式从“数据间关系”的角度上，为每个子系统（一组数据）定义了两类可能的产出，其一为结构型，即一组有关系的数据；其二为行为型，即一组有关系的逻辑。上述所谓产出的方式，既可以是指执行该子系统（而得到运算结果），也可以是指对该子系统重新结构（而最终达到某种外观表现）。

因此，GoF 模式本质上是说明：

即“结构（逻辑+数据）”这样的基本模型作用于数据间关系之后的产出。

这样看来，尽管我们讨论 GoF 模式时是面向（或基于）对象系统的，但其中的绝大多数模式与“对象”并没有必然关系。同样，即使我们将这些模式的应用泛化到纯粹的“数据的获取、展示与调度”这一层面，也仍然只是它——作为思维方式——的一种应用。

我们可以（也是可选地）将这一思维方式提升到系统层面，那么我们会发现，GoF 模式也可以是一种系统组织方式。这种情况下，系统中的工件并非是数据，而是各种应用与应用间的消息，前者可以理解为 s（f），后者则可以理解为 s(d)。

以此为起点，我们事实上是在将 GoF 思想延伸到系统设计的各个领域。例如著名的系统设计模式 MVC 以及常用的插件框架，就可以视作几种 GoF 模式应用于某种、某类数据以及某个领域的结果(48)。

————————————————————————

(1) 我这样分类的很大一部分原因在于：有必要将整个软件开发、程序设计阶段需要完成的工作，纳入到我们的讨论范畴。

(2) 数据结构与数据类型，前者是方法本身，后者是方法的表示。大多数情况下，也不妨将它们认为是同一个东西。

(3) 《结构程序设计》的第二篇中，作者霍尔（C.A.R Hoare，1980 年图灵奖获得者）采用数学定义的方式来说明数据类型：设已知有限个的数，称其每一个为基数，所有基数的集合则为基型；由已知类型——含已知结构型和已知基型——构成的类型叫做结构型。基型的构造成分唯一，也因此它被称为非结构型，它或是计算环境提供的，例如 WORD 与 BYTE；或是程序员定义的有限个数，例如枚举。综上，（数据的）结构是一个过程，或称之为结构化。

(4) 即所谓“算”是程序之表，“数”是程序之本。

(5) 这里我并不强调它是“必备条件”，因为有些计算机和计算机系统并不是由“人”来使用的，因此与它们交互的数据形式也不相同。关于这一点，应退回到本书第四章第一节中，在与“数据的提出”有关的话题中去讨论。

(6) 我们只假设了一种最简单的情况以便于后续讨论。现实是，需要考虑运算器与存储器之间的总线，以及指令与数据缓存、预取装置等，在整个数据的传输过程中的位宽，决定了这些数值的大小以及存取效率。

(7) 这样的表示法显然有非常非常多种可能性，而且也确实存在着好几种在用的方案。

(8) 地址标识与寻址问题是一个复杂而一体的问题，并非此处所说的“整型增量”这样简单。例如，在硬盘读写的寻址方面，就有多套完全独立的体系。这些体系与磁头、磁盘片以及高速马达的惯性等都有关系。地址标识与如何提高设备存取的效能，是计算机系统中“存储”相关的焦点问题。而在此处及后文中，我们仅从软件开发视角上讨论该问题的一个子集，并且事实上“顺序地址”也是软件开发中有关存储的一般抽象。

(9) 这里涉及表示的地址索引与存取大小之间的约定。实际上 x86 约定的地址是字节序的，因此实际上在最后的这个地址上只有一个 Byte 能被处理，而其他的位置将因为无法编码地址而溢出。

(10) 例如一个图形计算环境，就可以考虑以 RGB 为基础数据类型并建立起基于此的运算体系。

(11) 与此前讨论的地址索引问题类似，这个连续区域的实际大小也受限于可用的、能被编码的地址大小。

(12) 我们这里先讨论程序设计中一个狭义的数组概念，之后的讨论中会再进一步完善它。

(13) 理解为抽象概念中的“引申”。

(14) Struct type 一般译作“结构（类型）”，这里用“结构体”以与本书中讨论的、普遍含义上的“（数据）结构”区别开来。一些语言中，结构体也被称为记录，这同样也是数据库中“记录”称谓的源起——后面也将讨论到结构体与数据库之间的抽象关系。

(15) 本质上来说，我们是回到了对“数据的性质”的讨论，参考本书第四章第一节。

(16) 哈希也称为散列，它仅指这种映射关系而并不要求映射的结果是一个数值，例如用于加密的 Hash 过程，就是将源信息映射为加密指纹信息。

(17) 这些值的计算含义在这里不需要关注，现在只关注如何建立正确的“名/值”关系。

(18) 这是安全令牌、加密盾等实时密码发生装置的基本原理。

(19) 这里用“船”而不是“独木舟”（canoe/boat）的原因，仅出于中文在行文上的方便。

(20) 这也意味着有些知识是不能复制的。

(21) 我们将 ship 理解为行为的定义，而将 ship()理解为行为的能力——这出于程序与抽象表达上的需要，而与英文的惯例是不同的。一旦使用 ship()，则说明是指该“划船”作为一个系统的、整体的行为的实施过程。

(22) 我们还没有讨论到动态/静态语言这一分类方法，这里简单地从标识角度来定义它们，

即静态语言是指标识（例如“变量”）在程序执行前就已经完全预知的，动态语言允许在程序执行过程中新建它们。

(23) 这里是特指可以执行的机器指令。

(24) 这里的推论是相当重要的。综合上述的讨论，我们对于一个环境的依赖，事实上可以看成对“定义与查找（数据）”这样的行为的依赖。再进一步，这也就揭示了我们的计算环境之于“数与算”的抽象本质，以及将数据结构（包括以此为基础的种种算法）作为核心研究领域的真实原因。

(25) 对于“数”的支持，在动态或静态语言中并没有根本的不同，但其中的匿名函数在静态语言中实现起来要困难很多——不过这并非无法实现，例如 Delphi 2009 以后的版本。

(26) 与上述“匿名 XX”相对应的，我们称之为“具名”。

(27) 注意这里没有强调值与标识的绑定，这是因为存在变量的缘故。对于 Erlang 这类语言来说，所有“作为系统计算的前设”的值都必须是常量性质的，即：不可写且类型确定。

(28) 动态/静态绑定这个概念被用在很多的地方，我们这里只明确地阐述其中的一种形式。即一个无类型、无值的变量，可以动态绑定一个值来赋予它类型的概念，即后文的动态类型。除此之外，我们并没有讨论在静态语言中存在的一个有类型变量，在执行期进行绑定（赋值）的情况。

(29) 重写一个具名函数看起来是一个动态特性，但其实它并没有改变标识上的数据类型，所以不作为这里的动态重写来讨论。从另一个角度来说，具名函数的重写与 i++的性质是一样的。

(30) Undefined 在 JavaScript 中也是一个类型，但这是概念完备性的一种表现，并非我们这里讨论的数据结构。

(31) 也许有人会说“元素更少，则更易理解”，但并不全然如此。抽象概念过少的时候，我们的表达是倾向于重叠指代的，例如“船”，既是名词也作动词。在编程语言中，这通过 ship 与 ship()来表达，便已经具备了两个符号（与其组合）——这使得我们在抽象概念上，至少已经分出了数据与行为。

(32) 在《数据结构》中，这被称为“检索”，它与下一小节要讨论的“排序”是两个非常关键的“（计算）行为”。

(33) 这种结束有两种可能，其一是序列完全计算完毕，我们通常理解为“退出”；其二是序列进入一个无休止的循环过程，我们通常理解为“等待—就绪”。对于执行器来说，这个顺序的代码序列（静态文本的翻译结果）所代表的执行逻辑（动态的处理）既可能是操作系统所调度的 CPU 执行权限的入口，也可能是一个解释器的执行片段的起始——不幸的是两种情况都被称为 process，只是前者通常译为进程，后者则译为处理。

(34) 本例是 JavaScript 解释器引擎的一个实际解析结果。需要留意的是：不同的语言以及其翻译器在解析细节上可能会有不同，因而可能产生别的结果。

(35) 其一，图中的 1～4 为代码行号。其二，为方便绘制，图中省去了直接值结点，只留下表示数据依赖关系的标识。

(36) 内层的环境是外层的环境的一个参考，或称之为历史环境的一个映像。

(37) 在一个语言的具体执行环境中，还涉及边界对齐问题。

(38) 通过地址或名字。这是目前计算机发展中对这个问题的两个答案，但并非只有这两个可能答案。

(39) 所以我们看到最新的 Go 语言只支持所谓结构，而.NET 中的结构与对象之间的界线也相当模糊。

(40) 奥利—约翰·达尔（Ole-Johan Dah）与霍尔合写了该书的第三篇。达尔被称为面向对象之父，是 2002 年图灵奖得主。霍尔同时是该书第二篇的作者，是 1980 年图灵奖获得者。

(41) 这里的“结构”更适宜理解为动词。

(42) 例如不支持 RTTI（RunTime Type Information，运行期类型信息）、编译性的语言。

(43) 例如支持 RTTI 的语言。由于在运行期了解类的关系，因此类必须在 RTTI 中登记为一个类型，即类类型。

(44) 例如在函数式语言中实现类继承方式的面向对象系统。这时，类信息应该记录在一个构造过程（函数）中，并通过该过程向外公布。

(45) C 模型是 Delphi 的基本模型，并且由 C 至 B，是 Delphi 在抽象概念上从 Object 到 Component 进化的基本动力——亦即是说，“组件与组件库”是 Delphi 基于 RTTI 的成功实现。在作者的开源项目 Qomo 中，使用了 A 模型来实现“基于原型继承的类继承”，因为 JavaScript 的原型继承本身就实现了“构造过程”这一特性。

(46) GoF 对元类的叙述是没有的，但这不妨碍用 GoF 模式在语言去中实现元类继承。

(47) Adapter for Class、Interpreter 和 Template Method 的“范围”为“类”，表明它们可以是类上的行为，而并非是指它们“产生类”。

(48) 参考《程序员修炼之道——从小工到专家》之“29 它只是视图”中的“超越 GUI”一节。

第七章

应用开发基础

程序与算数之间的关系，早就有人描述过了。尼古拉斯·沃斯（Niklaus Wirth）说“算法+数据结构=程序”，并把这一论断用作书名，终成名句。其实这与《算数书》这一书名是同一个意思。古人与今人在类似事物上的、最接近本质的理解，其实是一样的。

从形式上说，特定计算机系统相关的“程序”是略有不同的——它们是为一个可计算系统编制特定序列的编码。这些编码的本意是可以控制机器的指令，只是为了让程序员和计算系统可以存在一致的理解，进而便于将程序员思维映射为计算系统的行为，我们才让这些编码变成了可供人们阅读的文本。

然而无论从算数的必要性，还是从计算机系统的必要性来看，我们都无法解释某些代码文本出现的原因，例如 Module。因为类似这样的一些抽象概念，既非可参与计算的单元，也非计算机系统可以理解的指令。

对这一问题的反思揭示了应用开发思想的出处。

所谓应用开发，并不首要关注对现实系统的抽象或基于该抽象的可计算性的讨论，而是面向“一个应用”整体的系统化思考。因而其思想的核心，便在于如何更加有效地解构再组织这一系统。最终，我们将这一系统化的产出称为软件，或更进一步地与它的市场行为联系起来，称之为软件产品。

第一节　应用开发的背景与成因

一

任何一个所谓的“应用”，首先必然是一个或一组“程序”。这意味着它总是能用此前讨论的技术来完成“程序的功能”，例如，我们总是可以将一个现实的问题抽象为对象系统，并面向该对象系统来实现程序逻辑。这一过程在我们此前的讨论中已经一再复述。

但是一旦我们开始讨论应用本身的问题，则必然涉及它的两种内在驱动力量：

□　所面向的是泛计算领域中的某个或某一类用户；

□ 所处理的是某一个独立领域中的单一问题或特定范围的问题集。

由于这里的“用户”是指非专业的计算机操作者，因此他们对程序的使用方法和维护方法决定了两类与“现实的问题抽象”相距甚远的需求：

□ 非功能性需求；

□ 非当前需求。

这些，就是应用开发所面临的全部问题的背景与焦点。

二

对于一个软件产品，我们需要将功能性需求与非功能性需求分开，但这并不是一件容易的事情。程序员总是按习惯来理解他们面对的事物，例如一个用户身份卡（User Id Card）首先是一个内存块，而不是一个抽象数据类型（ADT，Abstract Data Type）。那么基于这样的认识，什么才是对这个 User_IdCard 的功能性需求，什么又是其非功能性需求呢？

问题在于：User_IdCard 具有多个层面的数据信息，并且对于不同的程序员来说，对其不同层面的理解都是正确、可实施、可计算的。缘于此，在不同层面上对其功能性需求的定义也就不同。例如对于偏向操作系统的开发人员，User_IdCard 只有一个信息是有用的，即(1)：

这个长度值决定了程序如何分配和管理该数据的存储。因此这个开发人员会将他所理解的、以该长度值为核心的操作作为一个相对独立的部分区别出来，这些操作可能是如下一些函数或方法的运用(2)：

例如，一种可能的情况是：

这个函数与 User_IdCard 这种数据是有关的，但又与在最终界面上操作该应用软件的用户（例如户籍管理员张三）毫无关系。

我们还会面临对这个数据的第二类理解：如果这个数据是一系列性质的集合（例如结构体或对象），则每一个性质将对应于现实系统的数据。简单地说，例如：

其性质 Age 代表了现实系统中某个用户的年龄。与此相类似，User_IdCard 的每个性质都有

其确定意义，并有相应的行为。这些行为与 User_IdCard 有关，但仍然是与张三无关。例如：

到现在为止，随着计算机应用软件开发技术的发展，我们已经将上述两类对 User_IdCard 的理解放在一个语言的基础部分去实现了。大多数情况下，（高级的）计算编程语言通过抽象数据类型及其管理技术（例如对象、读写器、数据验证器(3)以及垃圾回收机制等）来将这些问题隔离在应用程序开发者的视野之外。

忽略了类似上述的问题之后，我们才触到了应用开发的边缘。(4)

三

对于张三来说，第一个真正有意义的功能是什么呢？

答案是：查看身份档案。

应用开发必须站在用户视角来看问题。张三的日常工作之一是查看身份档案，这也就是他的功能需求。这项需求可能需要分成三个实现过程：

□ 过程一：列举身份档案

□ 过程二：调出身份档案

□ 过程三：显示身份档案

我们需要明确，张三所需功能其实是第三项，即显示身份档案。这可以实现为在用户操作环境中的图形化界面，或是可以进行交互操作的文字框等。而过程一，其实是一个附带的需求，因为从用户操作流程上来讲，张三可能(5)需要先看到一个列表，然后才能选择到某个 User_IdCard。

过程二则是一个从计算机角度来看待问题的结果。也就是说，对于计算机的数据系统来说，“显示身份档案”是“调出身份档案”之后的一个后续行为。从计算机的角度来看问题，对于张三有意义的功能是第二项；但从张三的角度来说，他只关心第三项功能。

在实际开发中，我们讨论的将是类似如下的代码(6)：

在这个示例中，我们严格地将“程序员视角下的过程”与“用户操作视角下的过程”区隔开来：只有当 TShowCardForm.Create()并 ShowModal()发生时(7)，整个程序才是用户交互相关的。这一示例典型地将业务逻辑与用户交互逻辑分离开来，使得观察“什么是用户需求”成

为显示易见的事情。

对应于过程一和过程二所述的需求，就其功能的实现来看，其实包括了在三个层次（机器数据层、基础数据层和应用数据层）上对数据的所有操作。其中：

□ TIdCards 等类型的设计，以及 Card 作为可操作数据的内存分配与管理等，这些是与计算环境相关，针对机器数据层的设计；

□ Cards 基于有序编号（fromId）、Cards 是否存储在数据库或本地结构化文件中，以及针对上述存储的存取过程等，这些与系统相关的、内在的设定构成了基础数据层；

□ CurrentCard 作为单例的存在，TUserAction 以及它与 TModalResult 之间的关系等，这些是应用数据层的设定——基本上来说，将应用数据层整体去掉之后，基础数据层仍然足以支撑其他的用户需求与业务。

然而这些对于一个具体的操作人员（例如张三）来说，没有什么意义。最后，对于功能三，如果不考虑显示的具体效果的话，它只是在应用数据层上一个“界面交互”的实作而已。

但正是在这一点上，我们发现“用户需求”完整地影响了过程三的设计。

四

界面交互是否算作功能性需求？

在程序员看来，显示身份档案这个功能实现为一个“信息列表”就足够了，因为一个列表（list）足以显示全部信息(8)。但是该档案也可能被显示在触摸屏、移动终端或者桌面电脑等设备中，因而对于实际的应用环境来说，“如何显示身份档案”是该应用的一个实际功能。这样的功能可能用户没有提出但使用环境有此要求，更可能“显示成何种形式”以及“如何操作它”已经包含在用户提交的功能需求当中。更进一步地：

□ 用户可能要求同时显示指定个数的档案（列表）；

□ 用户可能要求显示档案信息的部分或全部（卡片或详情）；

□ 用户可能要求显示档案某一信息的局部或改换某种显示效果（显示头像）；

□ 用户可能要求显示档案能被持续更新（实时刷新）；

□ 用户可能要求某种或某类角色可以修改某些信息（操作员与主管）；

□ 用户可能要求使用某种装备来输入某些信息（OCR 接口）。

□ ……

在程序设计的阶段，我们可以将“信息列表”作为输出验证的手段，例如使用自动测试工具来验证接口（类似前面提到的 ListCard()与 LoadCard()函数）中数据返回的正确性。但是对应用开发来说，根据用户的实际需求而组织一个合理的展示与交互界面，是我们必须完成的功能。在某些情况下，应用程序并不要求有太独特的界面与交互，例如标准的 Windows 桌面程序就可以直接在 Visual C++中用 MFC 标准组件完成。因此基于操作系统或特定环境下的界面与交互，“就程序员的感觉来说”，更像是一种实现的必需，很难被视为是用户功能性需求的一部分。

即便如此，对于一个真正意义上的应用软件产品来说，我们还是必须将界面与交互作为功能性需求。这种产品特性事实上构成了用户对功能的认识，例如界面是用户操作产品功能的一个不可或缺的部分，因此是功能性的。与之相对，安装过程与操作该产品的功能没有直接关系(9)，因此是非功能性的。

考察这一分歧，我们还必须留意到界面与交互带来的细分领域问题：当它作为操作系统（以及其开发标准）的一部分提供时，它与程序员所在的计算机环境是同一领域的；而当它是构成某个独立领域的用户认识的一部分时（例如工业控制界面，或户籍管理系统特有的身份证扫描界面），它就是该领域下的一些领域知识。这一观点用在桌面应用或后台应用开发(10)，以及其他种种软件开发活动中也是适合的。

我需要进一步强调的是，程序员在界面开发中对界面绘制方式（或界面控件）的专业知识，在于“如何实现它”，而不是“如何让它表现得更符合用户需要”，后者是界面交互设计的职责(11)。举例来说，作为程序员，如何在窗体中正确地层叠多张 PNG 图是你所必须掌握的技术，而其中某一张图是否漂亮美观或符合客户的审美情趣，就不是你必备的程序开发技能(12)。

五

那么什么是非功能性需求呢？我将这一类需求理解为：即使这些需求完全不存在，也并不影响应用的主体功能的交付，那么这类需求就称为非功能性需求。例如安装、手册、升级维护工具等。(13)在我们的应用开发中，有大量的工作与技能都是与这些非功能性需求相关的。然而它们并非是不必要的，相反，掌握这些技能是成长为应用开发工程师所必需的。只是这些内容的思维方式与学习方法，与我们此前所讨论的“（严格意义上的）程序”完全不同。

综合上述讨论，应用开发中存在三个层次上的需求，如图 3-1 所示。

图 3-1　应用开发中三个层次上的需求

其中，必须完成的计算需求是通过系统分析，在机器数据层、基础数据层和应用数据层上所得出的功能性需求。独立领域的应用需求，是通过业务分析，在用户的业务领域中所得出的功能性需求。而第三个方面，产品需求则通常是非功能性的(14)。最后需要强调，有一部分非功能性需求可能同时也是非当前需求，例如版本(15)。

第二节　应用开发技术

一

对于应用开发中的功能性需求（计算需求与应用需求）来说，一切空间因素所致的复杂性，都可以通过组织形式来解决；一切时间因素所致的复杂性，都可以通过抽象模型来解决。(16)

第一类情况更倾向于工程师思维，在工程师看来，1+1 永远都等于 2，因此系统总是可以通过部件的持续增加来得到最终的结果。第二类情况则更倾向于管理者思维，在管理者看来，是否需要 2 就是一个问题。因此他们倾向于先完成 1，再讨论 2 的问题。于是他们只要求以某种形式或方法证明：系统具有可以演化至 2 的“可能性”。

因此在应用开发语言中也同时有着两个发展方向：一个是从“模块/单元”这一角度出发的软件复用，另一个是从“项目/工程”这一角度出发的工程组织。

二

化整为零是一种简单却不易行的软件开发策略。其关键在于，对于任何一个目标集来说，拆分的可能方案都趋向于无穷。因此我们为论证某一个拆分方案的正确性而付出的代价，往往要高于实施目标集本身。换言之，要讨论如何拆分是“最”好的，有时甚至不如不拆分。

所以事实上对于任何一个项目中的类、单元、子程序或子项目的抽取，我们在现在以及将来实施的都绝不是“最优解”，而是在所处阶段中相对“较可行”的一个方案而已。程序员在这一过程中往往是一个被动的实施者，而非这个“解”正确性如何的评估者。

总有一些规则是“较好的”或“较容易判断与实施的”，但如上所述，它们可能不是“最好的”与“最正确的”。了解这些规则，是渐进提高程序素质的方法之一，但并非某个具体项目的唯一判断标准。

三

图 3-2 展示了在稍早一些的应用开发语言中，从“代码的粒度”出发的抽象概念。

图 3-2　从“代码的粒度”出发的抽象概念

注 1：语句与行的不同，通常也被称为逻辑行与物理行概念上的不同。此外，有些语言是强制要求以物理行来表达“语句”这一概念的，即一行语句必须书写于一行代码中。

注 2：单元与模块除了称谓的不同，很多时候其抽象概念也并不完全相同或者互相覆盖，例如一个单元可以是（或不是）一个模块。我们这里只取在某些语言中、将模块特指为“一系列函数”的这一概念。

其中，单元或模块用于组织一系列函数，而一个应用(17)则是由单元或模块构成。在这样的体系中，“化整为零”的问题会变得相对简单，即如何有规则或有逻辑地将一堆函数组织成单元。这里的“规则”与“逻辑”阐述了组织法则的两个方向。

其一，我们可以设定一个简单的分类依据，使得位于同一个单元中的函数表现出一定的相似性。例如开发一个图形库，我们可以将与图形设备相关的功能放在 device 库中，将绘制功能放在 graph 库中，将渲染功能放在 render 库中，将与图形库无关但又与计算机基础环境相关的功能放在 base 库中，如此等等。最后，我们将一些杂乱无章的功能放在 misc 库中。请注意，这一切的分类依据是“功能的归属与使用者”。类似地，我们也可以依据数据的位置来建立分类依据。例如同样是这个图形库，我们可以将基础数据运算放在 bits 库中，并基于此建立关于图形运算的类型抽象库 types。接下来我们定义在不同设备上适用的数据结构，比如在存储设备中的种种文件格式 fileTypes、在内存中复制和运算的 dibs（设备无关位图，Device-Independent Bitmap）以及在某种具体显示设备中适用的 cudaTypes（CUDA，Compute Unified Device Architecture）等。这样依据数据（所处的）位置以及需要进行的计算进行分类，也便于将数据及其副本放在不同的环境下开发。

这一类的方案或试图交付一个可以被使用甚至被共用的功能集，或通过抽取不同层次（例如面向不同设备或不同场景）的代码，使之可以“或多或少”应用于不同的环境。与这个组织原则密不可分的一个问题是：如何使一个“单元/模块”向外公布它所具有的功能集。这形成了著名的“开放细节”与“公开功能但隐藏细节”之争(18)，如今后者已成为应用接口设计思想的主流，前者则部分地影响并推进了开放源代码这一思想。

但总的来说，这个组织法则只解决了一个应用中能被静态规则化的部分。无论如何，它无法满足“让程序运行起来”之后可能带来的种种变化。

其二，我们可以使得一个单元或多个单元中的函数存有某种逻辑关系。著名的“自顶向下程序设计”的思想，就处于这一组织法则所代表的方向上。例如：

□ 设有一个逻辑（X），功能是将 m 变换为 n，如图 3-3 所示；

图 3-3 基本功能：将 m 变换为 n

□ 由于 X 的规模巨大，我们将它分成三个逻辑步骤（顺序逻辑 1→2→3）来实现，如图 3-4 所示。

图 3-4　分解：三个步骤

虽然向下一层的分解并不限定各步骤之间的关系，但我们注意到此前讨论过的一个事实，即所有逻辑都可以被理解为顺序逻辑中的一个步骤。也就是说，步骤 2 依赖于步骤 1，步骤 3 依赖于步骤 2。随后我们继续向下一层分解，如图 3-5 所示。

图 3-5　持续分解：更多的步骤、逻辑或子系统

在图 3-5 中：

□　步骤 1 的子步骤 1.1 分成了 1.2 和 1.3 两个分支，最终以 1.3 为出口，传出数据 n'；

□　步骤 2 则被分解为三个子步骤的循环，并总是以步骤 2.3 为出口，传出数据 n"；

□　步骤 3 分解为两个顺序的子步骤，得到转换结果 n。

由此无论是针对顶层 X 这个逻辑，还是针对第二、三层各分解的逻辑，整体的逻辑关系都没有变化。所有的逻辑关系在函数与函数之间，以及在“一堆函数”与“另一堆函数”之间都可以被简单地抽象为“顺序依赖”。即使将这些单元/模块之间的关系映射到最终子系统的划分之上，这种逻辑关系也不会变化。例如从子系统划分上看：

□　子系统 1 被设计为“预处理器”（preProcessor）；

□　子系统 2 被设计为“分析器”（analyzer）或“过滤器”（filter）；

□　子系统 3 被设计为（下一阶段的）“数据供应器”（dataProvider）。

这三个子系统以及它们的组织关系就可以构成某种数据处理系统的、整体的、面向运行期的逻辑架构。

进一步地说，通常单元/模块之间的逻辑关系只是简单的依赖关系。这一关系足以支撑由结构化程序设计带来的计算需求，包括支持数据流转与逻辑执行。(19)

四

将数据或逻辑具有类似性质的代码放在一起，或者将逻辑之间存有关系的代码放在一起，这两种思路与面向对象用封装性来解决的问题是相类似的。一个对象，其属性是一系列（具有同类抽象性质的）相关数据，其方法是一系列与上述属性相关或相互间存有依赖的逻辑；对象的封装性决定了对象对外或对某个范围公布的接口。因此，一个对象或这个对象的类，其实有着与“单元”（unit）相同的抽象意义。

所以当面向对象出现之后，“一个单元中应该放多少个类”成了一个问题：如果一个单元可以放多个类，那么它与“库（library）(20)是用来容纳多个类的组织单元”这一抽象概念又重叠了。因此在早期面向对象语言的设计中，对这个问题的解释是含糊不清的，例如 Pascal/Delphi 允许在一个单元中放任何多个类，这导致“单元的组织原则”变得更加简单而含混：如果类之间相关或相似，就放在同一个单元吧。而晚一些的面向对象语言则较好地解释了这个问题：一个类即是一个单元/模块，或干脆进一步地取消了单元/模块概念。在具体表现上，例如 JAVA 或 C#就推荐在一个文件中存放一个（可以公开的）类，这个文件——或包含许多函数与数据的单元——将作为一个独立的组织单位存在。

图 3-6 说明了在面向对象的设计观念中，“类”其实是用来替代“function/unit/module…”等组织方式。不过在某些多范式语言中，例如 pascal 或 javascript，通常也允许这两类组织方式同时存在。但从本质上来说，这只是代码组织方式决定了一个“代码集”（source code package）在形式上有所不同，其内部的逻辑、数据以及更为底层的算法观念其实是大同小异的。

图 3-6　“类”的价值与局限：对传统组织方式的一种替代

随着系统规模的扩大，应用产品对“引入第三方代码”的需求也越来越明显。而“类”作为一个组织单位，其实是将逻辑和与其相关的数据、相关的抽象目标集中在一起发布，因此面向对象技术提供了相当高的可复用性。这一点正好迎合了上述需求（当然，换个角度也可以说，是需求推动了面向对象复用技术），因而如何在类的基础上进行更大规模的代码组织，成了一个重要的问题。

名字空间（命名空间）的出现与对象复用的思想有着密不可分的关系，但究其本质而言，名字空间下是否包含一个“类簇”（class cluster）却并不要紧。因为名字空间本身只是用于隔离不同的软件厂商、产品和子项目之间的代码，以及这些代码对外交付的接口。这种隔离需求本质上只是来自于交付物的名字冲突（例如 A 公司与 B 公司的代码库中都存在 TDynamicArray 类），而不是这些交付物的类型或结构抽象冲突。所以无论面向对象是否出现，在“函数/单元/函数库”这样的组织单位持续进化之后，必然会由于跨公司、跨领域的复用而出现与“名字空间”相类似的代码组织方式。

常见的名字空间的命名规范为：

例如：

名字空间可以由更复杂的分类规则构成。例如：

通常其具体规则是由不同的公司/产品/产品线来决定的。例如：

如同所有的代码组织形式一样，名字空间通常也与作用域相关。由此带来的效果，也就是它解决的需求是：A 公司与 B 公司代码库中的 TDynamicArray 类之所以存在“不同”，是因为它们所处的名字空间不同。这一点与用“单元内、单元外”来隔离标识符系统，以及用函数、语句甚至表达式的“作用域”来隔离标识符系统的性质是完全相同的。它们只是组织规模上的差异，而其抽象概念以及目的是一致的，只是自然地随着规模扩张而延伸罢了。

五

库，通常是一种代码或其对应产品构件的交付形式。这意味着库的表现形式是多样的，例如可以是源代码的一种组织形式，也可以是由源代码编译成的二进制结果；它既可能是一些单元或模块的、有或没有规则的包装或集合，也可能是将某些“类”有序集中在一起的一个泛指。一些常见的形式包括：

□ 在编译语言中，库可能用来组织代码，或翻译阶段中的中间代码。例如汇编语言中的库（.lib），就用于管理一些目标文件（.obj）的集合。

□ 在目标系统中，库可能用来指应用发布的一个组成部分。例如 Windows 中的动态链接库（.DLL）。

□ 在面向对象系统中，（类）库通常用来指一些类的集合，但并不表现这些类是以源码形式或是二进制形式提供。例如开源项目中的类库可能是源代码包，而.NET 的类库则是一些可以注册到系统中的、二进制的程序集（Assembly）。

□ 在应用环境例如操作系统中，库通常是指可在不同应用产品之间复用的运行期代码。例如 COM 库，或前面提到的动态链接库（.DLL）(21)。

综合上述以及更多的应用环境，“库”通常都不是指代码本身的组织，而是指它们的交付，包括最终交付之前的某种阶段。

“库”究竟以何种形式交付，以及包含何种内容交付，都取决于最终应用产品对未来交付形式的需求。换而言之，既然库是某个产品（在特定运行环境中）的构件，那么它必然满足该环境的要求和该产品的限制。例如：

□ 如果产品以平台依赖的二进制共享文件发布，那么库可能以编译阶段的中间文件，或者执行阶段的依赖模块的形式提供，例如 Pascal 的*.tpu 和*.bpl，或者 delphi 的*.bpk，C/C++的*.lib，以及.NET 的程序集、COM 组件等。

□ 如果产品设计为支持插件的，那么库将以动态链接库（DLL）的形式交付，例如 Windows

操作系统中的屏保（*.scr）、控制面板应用（*.cpl）、管理模块（*.msc）等，或者 Photoshop 中的特效插件（*.8bf），这些其实都是修改了文件扩展名的动态链接库。

□ 如果产品设计为动态资源的，那么库将以资源包的形式交付，例如 Android 开发中的 *-res.apk 文件，通常就是作为某个主应用程序的资源包来提供的，这样便于提供同一个应用程序的不同国家/地区的版本。

□ 如果产品设计为支持动态脚本的，那么库将以源代码包或脚本库的形式交付，例如 Mozilla Firefox 的*.xpi 插件，本身就是一个.zip 文件包，其内容则是一些 javascript 脚本及其依赖的资源。

“库”作为交付物以及最终产品的一个组成部件来提供，也意味着它是具有版本信息的(22)。这种版本信息与交付产品的版本相关，例如不能直接将用于 Mozilla Firefox 3 的插件直接应用于 Mozilla Firefox 6，而这一限制是作为版本信息（install.rdf 文件）包含在插件中一起交付的。

然而作为一个完整的交付物，应用产品可能是一个不带有任何“库”的独立程序（例如 Windows 环境中的.exe 文件），也可能带有更为丰富的产品信息，例如包括：

□ 产品依赖的操作系统或主程序版本，例如 Firefox 插件中的 install.rdf 文件；

□ 自身基于的运行库，例如 VC 的 Runtime 库 vcredist_x86.msi；

□ 与此前的发布版本的冲突或依赖，例如配置文件检查或修复；

□ 当前产品版本发布时附带的完整库、文件或资源，例如文件清单；

□ 当上述依赖缺失时，可能的获取渠道，例如 ActiveX 组件的 codebase；

□ 产品文档或相关信息，例如 readme.txt、帮助文件或在线帮助的网址链接。

通常应用程序开发的集成环境（或某些推荐性的套件、开发工具组合）会提供一系列的方式来生成上述的产品内容(23)，最后将这些文件打包并提供某种统一、便捷的安装方式。这些最终可以由“用户”在其机器环境下自主使用的交付物，在商业意义上称为“产品”，而在程序世界中，通常就称为“包”(24)。

综上所述，我们将库（library）与包（package）作为产品交付形式相关的两个组织方法如图 3-7 所示。

图 3-7 （产品的）交付形式相关的组织方式

第三节　开发视角下的工程问题

一

没有软件开发人员会用代码去“写一个模型”，反而是在系统的分析建模完成之后，用代码去描述上述“建模的结果”（即模型）。换言之，编程语言所描述的模型，其实只是建模结果的一个方面，这就如同书面语言、口头语言等都只是语言的一个方面。我们若是仅局限在一个方面去讨论模型本身，便又回到了盲人摸象的困局。

正因为模型描述的是现实目标的一个或多个侧象——这也如同编程语言中的数据并不能描述现实目标的整体一样，所以不同的人从不同的角度观察和理解现实目标时所得到的模型也就不一样。最终在建模过程中所讨论的仅是对系统的一个相对“清晰一些”的理解，所以在建模过程中也往往会有“识别（出某个模型）”这样的说法。

我们不可能也不必要描述系统的全像，这是必然的。因此到底要描述哪些方面，就成了一个实际问题。而这个问题的解，要追溯到问题产生之处。也就是说：谁需要建模？谁做建模？以及作为中介的模型，表达了谁与谁之间的沟通需求？总的来说，一个“建模者”所面临的需求主要来自这几个方面，如图 3-8 所示。

图 3-8　“建模者”所面临的主要需求

建模者并非一个单一职能的角色，因为整个系统将涉及多个不同领域、不同对话对象的建模。例如，在产品域看来，建模的目的是要保证产品是否实施或者实施过程是否可控，因而需要在这些问题上提供一个“可讨论的对象”；而在实施域看来，同样需要提供一个“可讨论对象”，以保证在用户需求和产品提供之间的一致。

类似这种“可讨论对象”有一个重要的前提，即建模者必须能够提供一种在讨论双方或多方之间理解一致的东西。关于这一点，在敏捷工程与传统工程中存在极大的分歧：传统工程认为应该通过“建模”来得到一个多方共同认识的抽象对象——即模型，并围绕这些模型来推动从决策到实施的全程；而敏捷工程则认为对于多方来说，最好的、能够无碍理解的东西是产品原型而非抽象模型，因此应该将工程中的多方全部纳入一个基于产品原型的、迭代实施的推进过程中，由具体的过程环节来决定沟通的细节。

二

但问题是：为什么不能将“原型”也理解为“模型”？为什么非得将画在图纸上、用几何线条描绘的东西才视为模型，并将这些东西的抽象与绘制过程才称为建模呢？建模的目的是得到一个“可讨论对象”，而（多数情况下）原型就是——在敏捷工程所设定的场景中——可

以被多方认可并予以讨论的对象。既然如此，那么再讨论“是做原型还是画图纸”的问题就没什么价值了。

从实施推进的角度来看，模型事实上允许我们将系统拆分成多个阶段，并尽早地预期了系统的每个阶段所依赖的（前一个阶段的、可能的）事实基础，因此模型具有可描述、可分析、可预期等性质(25)。而敏捷工程本质上是把决策域与产品域中的需求拉到了实施域中，就地决策与设计（产品），并将这一过程开放给用户，如图 3-9 所示。

图 3-9　敏捷工程并不能消灭上述需求

问题的总量并没有减少，但是这里的“可讨论对象”（即原型），变成了纯粹用于工程师与用户之间沟通的桥梁。这符合一些应用开发工程的现实场景，例如用户通常更关心产品特性是否能够得到满足，他们多数时候并不关注市场、产品或实施阶段等问题。因此敏捷（以及类似基于原型、快速迭代的）工程模型有着很强的实用性。

然而原型不能解决一切问题。在一定程度上来说，原型是轻量级的试错，而抽象模型则可以通过严密的论证与分析过程来得到决策依据。因此，原型与模型的适用领域也有所不同。原型适合于在参与者之间建立简单的、直观的、允许快速纠错的讨论对象，更适用于快速推进以及短期、逼近式的产品开发。而模型则适合于抽象出系统中方向性、支撑性、不易频繁变化的关键环节，在此基础上进行论证，以尽可能减少出错或预期风险，甚至更进一步地提供中长期的系统演进规划（例如产品版本、产品线等）。

三

应用开发的“集成环境”（IDE，例如 Eclipse IDE）不是给一个人用的，开发工具公司的“套件”（Suite，例如 VSTS，Visual Studio Team Suite）是为多种角色提供的。但这两点事实，大多数时候为工程人员所忽视。集成环境或套件都是基于工业生产的理念来提供的，这一理念认为：工业生产整体依赖于分工明确的产品过程。因此，开发人员需要代码文本的编辑环境，测试人员需要自动化测试与脚本驱动的工具，构建人员需要包、构件以及面向资源的管理界面，如此等等。

将这一切组织在一起的 Suite 或 IDE，就如同生产线的硬件环境。正如这个比喻所暗示的：在购买这个生产线的同时，也就意味着你需要这个生产线的过程模型与管理体系。大多数时候，我们的应用软件产品开发并不是工具用得不好，而是生产线组织与管理得不好。而这其中，“模型”的指导意义尤其被忽略。

MSF（Microsoft Solution Framework）描述了 VSTS 生产线背后的工程模型，其核心并非来自于技术实施，而是来自对管理 “项目/产品过程”所需要进行的权衡（图 3-10）(26)，这与“项目平衡三角（项目管理三要素）”所阐述的问题是一致的（图 3-11）：

图 3-10　项目要素之间的平衡三角

图 3-11　管理平衡三角

于是进一步地，MSF 提出了三种模型来解决对应的问题，如图 3-12 所示。

图 3-12　MSF 的三种模型与项目要素之间的关系

其中组队模型讨论项目创建时的团队、资源、沟通模式，过程模型讨论开发过程的控制与管理方法，应用模型则讨论产品定义、产品实现以及产品特性的管理。当这些模型映射到 Suite 或 IDE 中时，相关的要素就变成了类似如表 3-1 所示的关系(27)。

表 3-1　模型要素在理论框架与集成环境中的关系

这一类工具在“一致性”上的要求与“统一建模”、“统一过程”等思想同出一源。更确切地说，它们是“方法 + 过程 + 工具”这一传统工程模型的具体实践，即方法论决定了过程模型，工具是对方法论与过程模型的实现及其具体实施手段。

但问题在于：是要懂得使用工具，还是要懂得方法与过程？

四

集成环境或套件试图通过统一模型来建立整个团队对话、沟通、工作的一致化的界面，但这一过程其实并非依赖或只依赖特定的工具。

敏捷工程敌视那些强行植入开发工具的决策需求、管理需求以及产品需求(28)，进而否定“管理者、决策者以及产品相关角色”等在这一环境中的价值，并从“形式上”将这些角色推出团队，同时把产品定义与产品品质这些职责直接交给开发人员与用户。但是从根本上来说，它未能否定“过程模型”的价值。因此，大多数敏捷团队在弱化了管理与产品相关职责之后，仍无法摆脱软件工程过程（例如瀑布模型，以及基于瀑布模型的迭代）的约束(29)，只是他们有空间、有能力去选择更轻量、适用的工具来应付那些决策、管理与产品的需求。

除开这些因素，“敏捷”事实上是在探索用户与工程师进行有效沟通的最简方式。这种方式并不是说不要“模型”，敏捷工程师（也包括“不敏捷”的工程师）其实也试图为用户所描述的业务数据与业务过程进行建模(30)。这些工程师“乐于”做的一部分建模工作一般是源于业务的“数据与过程”，而这跟编程世界中的“算法与数据结构”有着近乎天然的映射关系(31)，只是他们将这些工作换了一门“手艺”来做罢了。例如在 UML 中，逻辑视图与实现视图构建的就是这两类模型。而他们“不喜欢”做另一些建模工作（例如完整的业务建模与

产品建模），只是因为这些内容与开发工作没法关联起来，因而在他们的工作中是不需要的。

将 VSTS 等单纯视为“开发工具”是一个根本性的错误。既然这是生产线，那么正确的做法应是明确工程师在生产线上的角色，并将适当的工作交付给他们。但现实是，源于那些错误的认识，一些团队过度地要求工程师在生产线中的职责，反而激化了他们对相应职责的抵制。而敏捷工程与敏捷开发方法，不过是在这样的抵制运动中所诞生的“革命性”产物。

五

我们回到问题本身：决策域和产品域为什么需要（它们特有的）模型？

决策者需要的是业务模型。所谓“业务”，其核心包括它的产品构成与收益形式。作为面向商业运作的决策者来说，成本的开销与控制方式以及收益的获得与分配方式是整个业务的关键。工程管理者关注的则是一个产品或一系列产品的生存周期。这表现为项目过程，以及与项目过程对应的产品模型。业务建模与产品建模的区别在于：前者并不描述产品的当前特性以及延展特性，而后者正好关注这些内容。

需要特别提及的是，工程管理者通常关注一个产品/项目过程的实施进展、阶段规划、品质保障、成本控制等内容，而对产品之于产品线的中长期特性关注得很少。但事实上工程管理者需要产品模型的真正原因是，他需要一个蓝图，以及这个蓝图在长远的演化“可能性”。对于一个具体的工程管理者（例如某个项目经理）来说，产品的下一个版本或下一个系列可能根本与他无关。不过即使在这种情况下，他也需要在某些决策中用到上述的“可能性”，将其作为趋势性判断的依据。

这些问题与城市建筑是类似的。业务模型相当于城市规划：东边要建行政区，西边建成科技园区，市中心以旅游和购物中心为主，而南边则进一步做旧城改造规划等。(32)产品模型，则相当于某些在建住宅小区的“XX 工程几期规划图”。这个图用在工程现场，每个施工者就知道他们在建造一个什么样的东西；用在售楼广告中，购买者就知道他们交钱买单的楼盘特点；用在小区周边规划中，大家就知道哪个地方该开个超市或为小区幼儿园留一个绿色通道；如此等等。

但是我们会发现——事实上我们做如上讨论与比拟的关键价值也在于此——尽管决策域与产品域确实需要基于模型来开展工作，但对于工程现场的施工者来说，这两类模型的意义并不大。例如你不能指望施工者因为楼房有商用与住宅之别就感到开心或沮丧，也不会因为大楼模型做得比别的工程现场精美，就能让施工进程更快更好(33)。

因此，业务模型与产品模型事实上——在大多数时候——无助于实施与评估实施的细节。这也意味着，在软件开发/工程中，这些模型所采用的语言（文字、图、幻灯片或实物样品）只需要与它们主要的沟通环境相匹配即可，不必非得得到用户和工程师（以及某个具体团队）的认可(34)。

第四节　应用程序设计语言的复杂性

一

回到最开始的问题。无论如何，工程师最终仍然要通过一个产品实现过程来满足用户的如下需求：

□　非功能性需求；

□　非当前需求。

这些是应用系统复杂性的主要构成。通常地，工程师会将它们映射为系统中的两个主要概念，即“约束”与“变更”，如图 3-13 所示。

图 3-13　面对两种需求背景，所提出的主要（实施）概念

至于实施这些概念的具体技术与方法，对于工程师来说也并不陌生。例如非功能性中的跨平台问题，在早期软件开发的实践中，就是通过伪编译指令，即编译期的约束来指定目标程序的运行平台；又例如命令行参数，即用运行期的约束来指定目标程序的特定配置。就目前来说，这些实现可以归纳为图 3-14 所示的一些思想与技术(35)。

图 3-14　实施层面的思想、技术与具体方法

接下来我们就来讨论这些稍为具体一些的“技术”，包括概念或方案，以及技术或手段。它们是相关的实施思想在具体的技术层面、在两个不同维度上的实现。总而言之，所谓的概念或方案，总是对我们对于技术或手段在具体实施过程中所得经验的总结、归纳以及抽象层面上的提升。

二

应用程序并不是直接运行在机器环境中的，它通常依赖操作系统，甚至是依赖操作系统的具体版本。严格来说，所谓“操作系统或其具体版本”所提供的，就是一个运行环境。基本的操作系统主要封装了对硬件系统的访问，这包括：

□　存储（内存、硬盘以及移动存储管理）；

□　标准 IO 设备（显示器、键盘、鼠标、打印机等）；

□ CPU 及其分配（进程与线程管理等）；

□ 网络、媒体、通信等。

除此之外，操作系统提供文件以及其他操作系统资源的访问。(36)总的来说，操作系统为上述内容提供的 API（应用程序接口，Application Programming Interface）可以理解为运行环境的界面。但多数应用程序不单单依赖这个环境，例如使用 Visual C++编译的应用程序还依赖 Microsoft 发布的 VC++运行库(37)。

应用程序语言中的许多编译指令、伪指令、条件编译指令，甚至配置信息文件等，都可以用来做环境设定以及依赖分析，简单的如汇编语言中著名的：

这个伪指令，复杂的则类似于 Linux 系统中的 Make 文件。

Make 文件也有很多种类，而且并不仅仅出现在 Linux 系统或某些面向 Linux 系统的语言中，事实上几乎所有稍具规模的应用程序语言与开发环境都有自己的 Make 程序和 Make 文本。为了提供更为复杂的 Make 功能，许多 Make 文本被设计为“另一种语言”，也带有逻辑、条件、变量等语言特性。但这些特性并不是相应“应用程序开发语言”的内容，而仅仅是用来对“这个程序的产生、分发等过程中的一些需求”加以规则化、程序化，因此我们通常也称之为 Make 脚本。

通过对规则的抽取、细化以及提供脚本化的支持，“越来越复杂地 Make”成了一个发展方向，几乎所有在“运行之前”能预料到的需求都可以通过下列技术来解决：

□ 再编译一次；

□ 分发不同的编译版本；

□ 提供独特的安装过程；

□ 特定的运行期环境依赖声明；

□ 调整程序的启动参数。

然而这种方案有三个致命的弱点(38)：其一，我们不可能预料到问题的全集；其二，我们为可能出现的问题所付出的代价，甚至要多于编写应用程序本身；其三，由于这一方案（从逻辑上来讲）容许无度的需求，因而可能因次要特性过于丰富而导致最终产品失控。与这样的运行环境相对照的是，应用容器通过“将应用限定在一个明确设定的环境中”有效地规避了上述问题(39)。正因为环境特性由容器来决定，因此应用程序的开发与使用都不会超出容器提供的特性集的范围。

应用容器通常明确地设定了应用的类型，例如 Web 应用容器或企业应用容器等。容器会根

据应用的需求来搭载相应的可选件，并通过统一、一致的接口提供给应用程序使用。这些选件与接口被标准化，变成应用容器整体方案的一部分。更进一步地，在这些范围确定、接口标准的方案周边，部署方案、优化方案、硬件配置方案、商业应用方案等渐渐地得以形成。

对应用与应用容器进行标准化，勾画了一个类似于“集装箱”(40)的王国，EJB（Enterprise JavaBeans）是这个方案的成功范例之一。EJB 的特性以及通用应用容器的规范，其本质上仍然是在“规则化”这一方向上，它们通过：

□ 编译或运行指示，例如注解（Annotation）、验证器（Validator）、配置文件；

□ 脚本化，例如事务脚本（Transaction Scripts）

等技术来实现“将非功能性需求限制在特定范围”。

三

对于一个应用程序来说，“插件”往往意味着它是一个动态链接库、静态链接库，或一个独立发布的脚本包。如前所述，库或包，只是一堆代码在组织上的措辞而已。因此插件的本质在于用组织形式来解决空间因素所致的复杂性(41)。

“插件”这一方案意味着插件的宿主（HOST）与插件本身（Plugins、Addons 或 Extensions 等）是分别发布的。这事实上与应用容器有一定的相似性，只是两者应付的问题集存有稍许差异而已：

□ 应用容器主要提供可配置的、可重新实现的环境，并通过标准化来提供跨业务领域的应用支持。尽管应用容器也有应用范围的限制，但通常该范围的边界更大。

□ 插件的宿主通常在产品、用户和功能这些方面基本确定，在技术方案上也通常是预先可确定的，因此“宿主+插件”更适宜作为一个具体应用产品的实现手段。

与应用容器类似，宿主也有跨平台的问题，例如 Firefox 浏览器。这个问题有两个解决方案：其一，如果产品的用户有能力在不同平台上编译并重新发布宿主(42)，或者宿主是针对特定平台开发而无需重新发布的，那么插件通常实现为与环境相关的二进制模块；其二，如果宿主有明确的跨平台需求，而其编译过程复杂或不宜公开编译方案与代码，或者用户没有能力或没有必要去区别与编译不同平台中的插件版本，那么插件通常实现为文本形式的脚本或预先编译为跨平台的中间代码/代码包。前者例如 Firefox 浏览器中扩展名为.xpi 的插件，后者例如 Java 的.class 文件。

插件技术是基于应用开发技术的“模块/单元”理念而来的。不同的是，宿主通常可以脱离部分或全部插件独立运行，而应用程序通常不能独立于模块/单元来完成编译。能否从“模块/单元”中去除强约束的依赖性，是识别能否代之以插件技术的可行方法。

如前所述，在面向对象的设计观念中，“类”替代了“function/unit/module…”等组织方式。因此在面向对象的设计中，“HOST 类 + 插件基类”是这一解决方案的基本设计。更进一步

地，由于名字空间用于应付更大规模的组织，并且通常也具有“作用域”限制的能力，所以使用“HOST 框架 +（插件的）名字空间”也是一种可行的实现。这在本质上与前一种设计没有不同，只是应用的规模更大，而且作用域范围控制更为灵活，类似安全性、账户等级限制等特性也往往更加容易实现。

但是基于面向对象的插件技术的问题在于：通常宿主与插件都基于相同的“面向对象语言”，因此具体的对象实现技术、二进制编译技术、对象封装技术等限制了插件的通用性。例如很难用 Delphi 开发一个“类”，并将它作为一个插件用在 Visual C++开发的宿主上。

这个问题的核心在于如何实现“组件复用”。这里的“组件”（Component）与此前的“模块”的区别在于：组件通常用于表达面向对象中的可复用对象/类，而模块通常用于表达结构化编程中的一个可复用件(43)。这两者所面临的问题是完全一致的，其解决方案也有着延伸关系，例如 Windows 中的 COM 复用与 DLL 复用其实是一脉相承的，而.NET 中的 Assembely 复用与 COM 复用也是类似的关系。

这通常涉及三种不同类型的复用（事实上划分它们的依据也并非唯一）：

□ 二进制复用，例如 COM；

□ 公共指令集复用，例如 JAVA 类；

□ 文本复用，例如脚本。

无论技术方案的提供商如何渲染这些技术，其最终的结果是一样的：开发人员终于可以用 Delphi 写一个 COM 组件，并让它运行在一个 Visual C++写的类工厂中；或者用任意的 Java 编译器编译一个.class，并分发给其他编译、运行环境下的宿主。当这一切成为现实(44)之后，组件技术的核心便集中在了宿主框架的设计、插件的加载机制，以及隔离宿主与插件之间或多个插件之间的安全性这几个方面(45)。

插件提供了通过剪裁功能来重新定义产品的不同版本（例如标准版、专业版）的能力。与此同时，正因为它在概念上继承自“模块/单元”，所以也是一个“化整为零”的典型方案，这使得在大型项目中实施持续集成具有了可能性。这其中的一个有趣的事实是：如果集成需要所有“模块/单元”的参与，那么集成的失败率就会在系统规模达到一定程度后出现级数性的增长；只有被集成的产品可以通过类似插件的机制，将单一部件的规模控制在一定范围内，持续集成——而不致阻碍整个 Team 的推进——才成为可能。

四

通过应用开发来“交付一个版本”，只是产品过程中的一个环节。只有当“产品”本身就是源代码包的时候，这个产品过程才变得跟开发人员息息相关，例如开发人员将 GitHub 或 ClearCase 视为版本管理工具，并将其上的某一个分支或基线作为一个“版本”。

然而从产品过程的全程来看，一个“版本”包含的内容更为丰富，上述“（开发人员所理解的）版本管理”只是产品过程的需求在开发环节的一个投射而已。总的来说，产品过程是一

个工程问题，而非一个开发环节的技术与工具问题。它的部分问题集，被开发商置入了集成开发工具，并交付给开发人员使用。这个“部分问题集”实际上包括需求的变化以及与此相关的、变化的实现过程，而这是目前对于这一问题的“几乎全部”理解。

然而事实上这并不完整。例如我们的集成开发工具以 Project 或 ProjectGroup 为关键词来管理一个项目，但在产品过程中却是一个 Product，或一个 ProductLine。这一抽象概念上的差别带来了极大的思维空间，即开发人员是否应当基于 Product/ProductLine 来组织开发活动并进行所谓的“版本管理”？换言之，在 IDE 中是否应该出现比 Project/ProjectGroup 更高层次的组织行为，以及相应的、代码中的关键字？

事实上，加入了 CompanyName、ProductName 的名字空间就已经有了类似的性质。然而这一切，与在 IDE 中对产品过程加以映射、组织、管理与维护还有相当大的距离。

五

现在，大多数程序员都可以写出一个具有典范意义的“Hello World”程序(46)：

同时也会真正地、从个人意识中忘掉这样一个程序的含义以及需求不过是：

```
print("Hello World");
```

如果仅以这段程序而言，用户的需求仅用上述代码即可实现。而 Java 或其他一些应用程序开发语言则在这样的代码中加入了更多的概念，诸如：

- 类与对象等，例如：class HellWorld
- 名字空间、导入导出等，例如：System.out 以及隐式地导入 System
- 方法、属性、事件等，例如：print
- 类方法、类静态成员等，例如：static
- 引用、值与无值，以及基本类型系统等，例如：void
- 可见性、作用域等，例如：public
- 字符串值、字符串类等，例如："Hello World"与 String args[]

□ 应用入口与运行环境约束等，例如：void main()

除了这些概念(47)，在具体的开发环境中还会有容器、包、配置脚本、服务、模型、验证器、指示字、伪指令、分发、部署、版本容器、基线等概念，以应付不同角色的需求。

当一门语言从“实现程序功能”变成要“实现产品需求”时，其内部的语言设计思想也渐渐地变得不遵守“算法 + 数据结构 = 程序”这一经典法则。回顾“第五章语言及其面临的系统”，我们可以将这一切的变化以及可预期的、语言进化的方向都归结为：

通过在程序组织上的结构化来解决规模问题。

————————————————————————

(1) 这里基于 Delphi 语法惯例，用前缀字符“T”来表示数据类型，而 SizeOf()函数用于取数据类型所需存储的大小。

(2) 该例援引的是 Delphi 中的一些内存管理函数，在不同的语言或平台中可能存在差异。

(3) 这里的验证器指的是类似 JAVA 中的通过注解来进行数据验证的技术。

(4) “如何屏蔽底层细节”也是应用开发技术的一部分，但对大多数应用开发语言/环境来说，这些都是内建机制。

(5) 这只是“可能”，因为张三也可以有许多不同的方式来接触到这个 User_IdCard，并最终显示它。

(6) 对于现实的户籍管理系统来说，这可能算不得“好的”实现过程，因为它混杂了过程式与面向对象编程风格，并且对于多线程/多界面操作来说存在隐患。

(7) 这里使用了 Delphi 的一些技巧：（1）TShowCardForm()在关闭时是可以自动释放的；（2）窗体可以向调用过程返回一个值以表明用户所做的界面操作；（3）使用 ShowModal()方式打开的窗体能阻止该应用中其他窗体的操作，即这种情况下的用户界面是独占的。

(8) 想象一下把 xml 文件直接展示在文本编辑器或 WEB 浏览器中。

(9) 张三也许并不知道“户籍管理系统”如何安装到他工作的机器上，但这并不妨碍他使用这个系统。

(10) 可以在 PC 机、移动设备以及其他的各种环境中找到“桌面”与“后台”的不同划分。一种不太严格的划分方式是：后台程序总是以控制台或类似方式与操作者交互，而桌面程序则采用种种“友好的”交互方式。

(11) 这往往被归入产品的设计特性。Marty Cagan（网景副总裁、eBay 产品管理及设计高级副总裁）甚至将设计与功能区分开来，将设计特性作为产品属性而非具体的功能属性。参见《程序员》2011 年之“Marty Cagan 谈产品系列”。

(12) 你当然可以具有这样的技能，一专多能或者无所不能并非坏事，但在这里我们只讨论属于程序员的那个部分。

(13) 有一类软件叫“绿色软件”，它有一个特定的名称叫“Portable Software”，通常是某个应用产品的定制版。比较绿色软件与相应产品的发布包，其中被删减掉的部分基本上就可以称为非功能性需求。

(14) 这些狭义的、刻意与其他类型的需求区别开来的产品需求当然也表现为具体的功能，这里的“非功能性”是针对目标用户而言。例如产品互联网产品中存在如何运营、营利的问题，这类运营需求往往不是目标用户所关心的，而是互联网企业对于该产品的需求。

(15) 在这一视角下，“版本”这个概念是相当混杂的，它包括功能性需求、非功能性的需求，以及产品和产品线的需求。在下一节中我们会再次谈到：在“应用开发”这一领域中所言的版本，仅是这个概念中的一部分。

(16) 这里的空间与时间因素，是基于在架构分篇（第四篇）中对目标属性的分类方式来讨论的：所谓空间需求，是指它们可以通过组成部件的增减来解决；所谓时间需求，是指它们可以划分为多个时间阶段来实施。

(17) 早期的应用开发语言也直接将应用称为“程序”（program）。

(18) 参见《人月神话》中“关于信息隐藏，Parnas 是正确的，我是错误的”小节，以及 David Parnas 关于信息隐蔽理论的著名论文：《论将系统分解为模块的准则》、《设计易于扩展和收缩的软件》和《复杂系统的模块化架构》。

(19) DFD（DataFlow Diagram，数据流图）通常用于解释在上述执行过程中 m-n 之间的转换关系不变（即数据单一入口与单一出口）。但在本例中，它亦用于解释自顶向下过程中的逻辑关系不变，整体保持着顺序执行关系。这一过程是抽象概念——从程序语言中逻辑的结构化，到应用系统中组织的模块化——的延伸。

(20) 这里指的是对象库（object library）或类库（class library），而普遍意义上的“库”是下一小节讨论的重点。

(21) 动态链接库也有动态加载和静态加载两种形式，前者是用户代码可以使用 loadLibrary() 等函数将该库装载到应用环境中，后者是指在编译期由编译器决定的、在应用运行的初始化阶段由操作系统负责装载的库。在 Windows 环境中，一个库是用于静态加载或动态加载，通常是由应用决定的——这也意味着它的发布与依赖也由应用来决定。

(22) 版本也是导致“DLL 地狱”（DLL Hell）问题的根源之一。所谓“DLL 地狱”，是指在同一执行环境的不同软件产品中，由于使用了同名但版本不同的 DLL 而导致的冲突。这一问题并

非 Windows DLL 或 Linux Library 所特有。不同的库，在解决这一问题的策略上不尽相同，但大体上都是以“声明版本依赖关系”为基本思路。

(23) 这些内容可能被称为“库”，也可能被称为“构件”，还可能被称为“依赖项”，如此等等。

(24) 例如对象库或类库的交付物称为“类包”（class package），可执行应用及其完整交付称为“安装包”（install package），而 linux/debian 环境下一个产品或产品系列的发布称为“debian package”。

(25) 正因为如此，上一节一开始就讲到：它适于解决时间因素所致的复杂性。

(26) 引自“MSF Process Model v. 3.1”, Microsoft Solutions Framework White Paper。

(27) 这种对应关系并非是边界分明的，事实上 IDE 中的同一个功能可能应对的是不同模型中的问题或问题集。

(28) 导致这一局面的部分的、且并非关键的原因在于：一方面，开发工具中使用 Project、ProjectGroup 等关键字来应对项目规模的扩大，而这一问题延伸到工程模型后所得到的抽象概念却是 Product、Version 以及 ProductLine 等；另一方面，工程工具与开发工具的提供商试图将这些合而为一，但形式上的统一未能弥补概念与领域上差异。

(29) 这进一步导致了团队无法摆脱既有的公司组织与管理结构，于是实施敏捷变得在组织层面上既说不通也行不通。

(30) 例如，正如此前所讨论的，敏捷中的原型事实上也是敏捷工程师与用户之间沟通的模型。

(31) 你可以想象成开发过程中的“定义数据结构和写函数声明”，以及形式化的流程图与部分逻辑代码。

(32) 在地图上描绘类似这样的一个规划布局，然后试着将一个人工作生活的行经路线画出来，标上每天、每周、每月重复路线的次数作为权值，于是你就可以论证“为什么我们所在城市的交通这样拥堵”了。

(33) 这里特别地忽略了一些细节，例如某种类型的楼（或软件开发产品）更难建造，或某个地区的楼（或业务方向）更受领导重视等。这些的确影响到具体团队、人员的选择，但它们并非这里讨论的“模型作为沟通工具的价值”。

(34) 一个设计或架构模型“要让所有人都理解”是相当荒谬的，而统一建模语言——请注意，我这里并不是指字面上的 UML，而是指“统一（某些东西）”这种思想——便是这类荒谬想法的具体实践。

(35) 对于功能性需求来说，部分（尤其是时间与空间的区划）可参见本章第二节：应用开

发技术。

(36) 并不是所有的操作系统都提供文件访问支持，这取决于该系统对于“操作系统（的）资源”的定义。例如 Windows 系统中可以将注册表理解为资源，而 Linux 系统中就没注册表。

(37) 对于 Microsoft 产品来说，操作系统中通常会带有该运行库的一个或多个版本，但某些情况下，仍然需要应用程序自己来管理这个库的安装与维护。

(38) 这些弱点的表现之一是：Linux 下繁杂的产品生态（例如不同的编译、发布与维护版本），以及非功能性需求的实现过于复杂（例如安装）。

(39) 运行环境与应用容器两种方案，在解决问题时所处的层面是不同的。前者是从系统的视角出发来限制性地提供应用的背景，后者是从领域的视角出发来提供对应用范围的假设。

(40) 参见《集装箱改变世界》，Marc Levinson 著。

(41) 这仍然是本章“第二节　应用开发技术”所述观念的具体实践。

(42) 例如 Apache 采用“WWW 服务器 + 扩展库（动态链接库形式）”的形式来交付，一方面是考虑到二进制模块的执行效率，另一方面也因为编译发布是部署人员应有的技能。通常来说，Linux 用户会更多地面临这一类情况。

(43) 并不一定是源代码中的一个单元（unit），而是指在指定的应用开发环境中，由语言或平台决定的“可复用单位”。

(44) 我的意思是组件复用成为应用开发环境的一个内置技术，例如 Java beans。

(45) 尽管并不明显，但事实上这种隔离通常与消息机制有关。无论是从“消息”这一概念的历史还是从它在应用系统中的使用来说，它都是“体系性”相当明显的一种技术。因此本书将在“第九章系统的基本组织方法与原理”中去讨论它。

(46) 引自：http://www2.latech.edu/～acm/helloworld/java.html

(47) 其大部分与“面向对象”这一语言范式有关，部分则出于应用环境、语法习惯等在语言的具体设计上的选择。

第八章　系统的基础部件

解决“不确定”问题，需要首先将其背景置入到“确定”与“不确定”得以出现的本质原因中去。

正是数据不确定带来了对观察者的限制，进而这种限制带来了所谓数据连续或非连续这样的特性。但反过来说，如果数据是确定的，我们将不必限制观察者，也不必讨论系统是并发的还是串行的问题。

在一个足够小的生存周期中，我们可以做到数据确定；在可做到数据确定的前提下，我们将数据的生存周期扩展到足够大，则可以做到数据连续。在现实系统中，前者影响的是实时性，后者影响的是并发性。也就是说，高实时与高并发是最难兼得的系统特性，因为高实时意味着数据的生存周期小，也就意味着并发中面临数据失效（即连续性的背景——生存周期，达不到足够大）的可能性大。

大而化之，从结构的层面来讲，其中是可以有很多种解释的。

第一节　分布

一

聪明的曹冲称出了大象的重量。此前我们仅仅将这一思想归纳为“通过某种系统将大象的重量映射为石头的重量”，但这还远远不够。因为我们只触碰到了这个问题的一个解——映射，而“为什么需要映射”才真正是问题的本身。

曹冲的高明之处在于，他认识到“不能称象”的本质原因是：大象不能被分割。“不能分割”才是灾难之源。因此，如果系统如我们此前所讨论的，是“通过在程序组织上的结构化来解决规模问题”的一种策略，那么程序所解决的问题集“能否分割”以及“如何正确地分割”，就是所有系统问题的核心所在。

对此，曹冲称象的故事提出了一种可能的解：如果“被运算对象”是不可分割的，那么我们可以将它映射为可分割的对象。但即使这个解总是存在的，我们也只是看到了问题的一半。因为在曹冲称象的故事中，我们忽略了一个非常重要的事物：秤。

秤，实质上就是一个数据处理系统：

□　其一，它具备一个数据处理系统的两个基本要素：处理信息与反馈信息，例如称一块肉，并反馈结果：二斤六两。

□　其二，它存有一个基本限制：能够处理的信息边界，例如只能称重 100 斤。

也正是因为秤的数据处理能力有限，我们才称不出大象的重量。所以整个“称象问题”既可以看做是“象太大”，也可以看做是“秤太小”。

我们的计算机理论是基于这样一个事实，即计算系统的本质就是“算数”。而“被运算对象”的分割只是解决了其中“数的问题”。因此在我们将逻辑上的计算系统映射为一个实际实现，例如“称象”或者“计算机”时，我们也可以尝试去解决“算的问题”。

换言之，系统应付规模问题的总法则只有两个：

运算能力的分布，以及运算对象的分布。

二

但什么是“分布”呢？

分布并不等于分割。“分割”是指一个问题集（无论是运算能力还是运算对象）能否被切分，例如前面一再提及的大象就是不可被分割的。而“分布”，指的是分割的结果能否被各个独立地加以处理。因此，我们说大象映射为石头之后具有了“（数据的）可分布性”，并不仅仅是因为在形式上进行了分割，还因为这些石头能被逐一称重。所以说：

“可拆分”与拆分的结果“可处理”，这两个特性在“分布”中缺一不可。

与一把秤相类似，一个函数实质上也是一个数据处理系统：

□ 其一，很明显，它能“处理数据”(1)并反馈一个返回值；

□ 其二，函数能处理的数据也有类型、边界以及运算总量的限制。

就我们通常使用的、单处理器的个人计算机而言，“所有软件构成的全集”所提供的功能总和可以被理解为“一个函数”（设为函数 F）。因为从计算机通电运行开始，系统开始了唯一一个程序入口与处理过程：

□ 步骤一：进行系统自检、BIOS 预设等常规的、硬件系统自有的处理程序；

□ 步骤二：尝试按照约定次序加载移动存储、外部存储等设备中的处理程序(2)；

□ 步骤三：将系统的控制权转移给步骤二中找到的处理程序（的入口）。

请注意，在上述这个过程以及其后的全过程中，处理器（CPU）的处理其实是在单一的时间序列中进行的。而我们的操作系统之所以能同时运行多个程序（例如 Windows 的资源管理器与记事本），以及在后台与前台运行不同的服务与应用程序等，是操作系统：

□ 将“所有软件所提供的功能总和”，即是我们上面假定的函数 F，分成了多个函数 F0，…，

Fn，计为 F′；

□ 将 F′理解为进程的入口，并假定 F′可以再分成多个函数 F′0，…，F'n，计为 F″；

□ 将 F″理解为线程的入口，并假定 F″可以再分成……

我们的操作系统（或某个硬件环境下的软件系统）无非是在对需要运算的总量进行拆分，并尝试将这些拆分结果分布在“不同的逻辑单元”(3)中进行处理。

这一计算模型在单处理器时代被称为“分时处理”，即通过任务调度，将单一处理器的处理能力分配给不同的函数。这些函数在宏观层面上是同时运行的进程、线程等执行体，即并行；而在微观层面上是分时运行的、单处理器的时间序列下的一个时间片，即串行。

回顾分时处理模型，其核心仍然基于“顺序机器”这一基本假设。与此相应地，其基本计算逻辑也是基于结构化程序设计观念，即可以将分支逻辑与循环逻辑统一为顺序逻辑的一个部分。进一步地，也可以将函数理解为一个子函数序列的连续运算。再进一步地，可以说：

如果子函数可以分布，则整个系统是可以分布的。

三

在“进程－线程”模型中，如果将进程想象为一个函数（例如 main()），那么操作系统将认为“各个进程所对应的函数 main()”之间是可以分布的，也就是可拆分，并且每个拆分单元都是可处理的。因此为了达到这一效果，每个进程配置的“资源”也都是一样的，例如各自拥有显示器、硬盘、内存（地址空间）、键盘等虚拟设备。大多数情况下，在操作系统层面屏蔽了这个事实：上述的设备是硬件唯一的，每个独立进程只是持有了这个设备的一个“（可操作的）映像”而已。

无论如何，这些构成了我们的操作系统“能分时处理”的事实。随后工业界便一股脑地将同样的问题与同样的解决方案套用到“多处理器（多核）”计算机中去，认为在这样的计算系统中，无非是将“能分时处理的”那些函数放在了不同的处理器中而已。

然而这一切的基础并不牢靠：子函数真的是可以分布的吗？

对于一个函数而言，可拆分总是必然的。这是基于顺序执行的一个简单推理：若一个函数总是由顺序、分支与循环逻辑构成，且分支与循环总是可以被视为顺序逻辑的一个步骤，则函数必然可以拆分成多个顺序逻辑的步骤。

但拆分的结果（设为函数 A 与函数 B）是否都是可处理的呢？既然函数 A 与函数 B 是拆分自同一时序下的两个逻辑，这涉及两个关键问题：

□ 其一，若函数的逻辑本身不依赖该时序，则可以处理；

□ 其二，若函数 B 所处理的数据，在时序上不依赖函数 A 的处理结果，则函数 B 可以处理，反之亦然。

这两个问题的反例可以分别被称为：

□ 逻辑依赖时序，例如函数 B 用于计算函数 A 的执行时长，则函数 B 必须在函数 A 逻辑结束之后执行；

□ 数据依赖时序，例如函数 B 用于计算函数 A 的结果的倍数，则函数 B 必须使用函数 A 的结果数据。

它们准确地说就是“（逻辑或数据的）时序依赖”，即在时间维度下不可分解。这预示着我们的“函数”总存在无法拆分的可能。换言之，必然存在无法通过“分布（或组织的结构化）”来解决的规模/复杂性系统问题。

四

所幸在不那么学术的环境中，我们的应用系统只需要解决问题的部分，而非全部。(4)所以我们面临的往往是第三类问题，即对于函数 A 与函数 B 来说：

□ 若函数在拆分时已经持有了所需处理数据的全集，则总是可以处理。例如：函数 B 计算 x*3，而函数 A 计算 x*x 的值。

所以就现在讨论的问题（的子集）来说，函数是否可以再被拆分其实受限于它所处理的数据是否可以分布。这也存在两类问题：其一，函数 A 与函数 B 所持有的能否是 x 的不同映像，即 x 可否存在各自独立的多个数据映像；其二，函数 A 与函数 B 是否能够持有所需处理的数据全集。

这在逻辑上是有解的。曹冲称象的故事提出了一种可能的解：如果“被运算对象”是不可分割的，那么我们可以将它映射为可分割的对象。所以在逻辑上，函数 A 与函数 B 总是可以持有

□ 所需处理的数据全集，或

□ 其各个部分的映射（的部分或全部），或

□ 其整体的单一映像。

而我们最终要解决的，只是存放这些数据、映射或映像的方法，即存储问题。

五

如前所述，函数的拆分与数据的拆分有一定的关系。总的来看，若函数 A 与函数 B 之间没有“时序依赖”，则函数 A 与函数 B 能否拆分取决于它们所处理的数据是否能拆分或复制（映

像）。

根据函数本身的结构化性质，当某个函数拆分成函数 A 与函数 B 时，必然是三种逻辑结构所映射的关系。进一步地，它们对数据拆分的需要也各有不同。

其一，顺序结构意味着函数 A 与函数 B 可以使用数据的映射（的部分或全部）。例如下面的代码：

foo()函数持有了 a、b、c 三个数据的全集，并且我们假设——事实上我们是特意这样构造的——函数的两个子步骤（代码 6、7 行）之间没有时序依赖。那么我们可以将 a、b、c 映射为两个数据，并在各自的子函数中使用它们：

在示例 1 中使用“var”声明的数据总量被拆分成示例 2 中的两个对象，由于示例 1 中的两个步骤之间是顺序关系，因此它们可以分别使用示例 2 中的两个 data，即“数据总量的映射”的部分(5)。

其二，分支结构（以及多重分支）类似于顺序结构，函数 A 与函数 B 可以使用数据的映射（的部分或全部），但是函数 A 与函数 B 相对于条件判断逻辑都存在“（逻辑的）时序依赖”。例如：

在这个 foo()示例中，函数的两个分支之间是没有时序依赖的，但是它们都必须在 x 这个逻辑之后执行。由于 x 相对于两个分支不存在——或可以不存在——数据依赖关系，因此两个分支也可以持有各自的 data。例如：

请注意一个有趣的事实：示例 4 所使用的 foo_1()和 foo_2()，与示例 2 中是完全一致的。这意味着这两个逻辑以及相关的数据，与其外在的其他逻辑无关。这体现了它们的可分布性，即可拆分与可处理。

在示例 4 中，对于 foo()函数来讲，foo_1()与 foo_2()相对于 x 这个逻辑都存在“逻辑上的”时序依赖，但它们之间以及它们之于 x 的数据，都不存在依赖。

其三，循环结构意味着函数 A 与函数 B 使用数据全集，或其整体的单一映像。例如：

首先，一种错误的理解在于将 5、6 两行代码视作不存在依赖的两个子过程，进而做这样的

处理：

尽管在这样的逻辑中，foo_1()与 foo_2()是可以持有数据的部分或部分映像的。但这与我们在这里讨论“循环逻辑”的初衷是相背离的。

我们事实上是在讨论将“一个具有循环逻辑性质的函数”拆分为多个子函数的情况。我们的目标是找到与“循环逻辑”这一性质相关的数据处理方案，而非将循环逻辑映射为多个却无视该逻辑之于数据的关系。在上述方案中，循环之于数据的性质是没有丝毫变化的。

将“循环逻辑本身”拆分开来，其基本含义是循环项次的展开。也就是说，我们能够将 100 次循环变成两个 50 次，或者 100 个 1 次。我们讨论的是这 50 次或 1 次中的数据之于“循环项次的展开”的逻辑间的关系。然而关于这一问题的答案是简单的：每一个循环项次，都必然面临数据的全集，或其全集的映像。因为 5、6 两行代码在时间上——可以理解为在一个时间区段中——是关联的，而“循环项次的展开”只是将时间区段趋向无限小的分隔，而并没有将上述这一关联关系解构。

以函数式语言的处理为例，我们可以将上述逻辑变成一个基于函数参数界面的递归，例如：

我们应该注意到，仅以“循环逻辑的展开”而言，函数 foo_x()的任意一个实例都只依赖调用界面上的 i 值。而这个 i 值是一个循环过程中的中间值，或是一个传入的确值，都是与这个函数无关的。因此，任意递归函数的单一实例，对于“循环逻辑”都是透明的。

然而再观察上述的示例 7，我们发现函数 foo_x()的任意一个实例，无论它仅是一个单次递归，或是分布到其他计算环境中的一个迭代区段，它都必将面临整个数据全集：

一种较好的、较可行的方案是将这个数据全集也放在函数的参数界面上(6)。例如：

这样带来的结果是：foo_x()的执行可以被分布，但其“所有分布（的各个服务之间）”存在着逻辑之于数据全集的关联。在现实中，这一分布带来了逻辑向计算系统迁移的可能性，即一个大的循环过程可以分布在多个计算系统中完成，因而仍然是非常重要的大型系统下的分布解决方案。

但是整个循环逻辑与其占用的时间区段的总量并没有变化。

六

对于任意一个大型复杂系统来说，如果它能被拆分的话，我们拆分的模式无非上述三种。而且这种从逻辑的视角进行的拆分活动，最终也会表现为数据的拆分，例如子系统与子系统各自的数据库。更进一步地，如果我们不考虑这些子系统之间的数据关系的话，整个系统将是可持续分布、并行计算的。

虽然从我们目前的分析来看，这一切是成立的，但是现实中的系统往往并不这样简单。其中的问题之一，在于“数据并不总是可以拆分”。例如一个简单的统计日志功能，其原始的需求如下：分析某个日志文件，统计日志的总行数。

缘于这个日志文件过大，我们对日志文件进行了数据拆分：每 100M 数据分成一个文件。这样，我们在整个逻辑上就可以理解为：对于文件 log1.txt 使用逻辑 A1，对于文件 log2.txt 使用逻辑 A2，如此等等；对于整个逻辑，我们将对 A1，…，An 返回值求和(7)。

但是 log1.txt，log2.txt，…，logN.txt 这些文件之间却存在着关系。由于数据按 100M 分段，而换行并不总是发生在分段的边界之上，所以 log1.txt 末尾可能有“半行”属于 log2.txt 的第一行……类似这样的关系，使得我们在拆分 log.txt 这个数据总量时存在一些实际限制。

这些数据方面的限制，通常是通过逻辑来弥补的。我们总是试图让“数据拆分”这一活动更“简单”，或更“规则”。简单，意味着分布它的成本很低，例如“数据按 100M 分段”的分布成本仅仅是物理存储上的限制；规则，意味着处理它的成本很低，例如“数据总是在换行边界上分段”，那么处理它的程序将非常容易写，并且判断逻辑会少很多，执行效率也就高很多。

但如同算法时间与空间复杂度难于平衡一样，分布成本低通常处理成本就高，反之亦然。也就是说，对于既已存在的数据集来说(8)，简单而又规则地拆分是两难之事。

第二节　依赖

一

我们提到过“（逻辑或数据的）时序依赖”，是在时间维度下不可分解的。这意味着某些逻辑不可分布，而另一些情况下，某些数据也不可分布。为此，我们讨论了一种可能的解：数据不可分布时通常会使逻辑趋向于复杂，例如需“判断 100M 数据的边界上是否存在换行”等。但这样的思路让我们陷入了困局：逻辑的时序依赖是否有解呢？

“挖坑、栽树”是一个（典型的、纯粹的、逻辑的）时序依赖活动。首先，这个系统中涉及的“坑”与“树”是不同的数据，相互间没有必然的依赖关系。其次，我们的的确确无法在时序上把它的逻辑倒置过来：无论是一个人来做，还是一群人来做这件事情，总得等挖完坑后才能栽树，这时多个团队或者 CPU 的多核并没有起到作用。

但只是当我们将“挖坑、栽树”过程理解为“在位置 A 上挖坑，并在位置 A 上种树”时，“位置 A”就成了这两个步骤的数据依赖。一旦去掉这个数据依赖，“挖坑”与“栽树”这两个行为本质上其实是没有依赖的，例如在“在位置 A 上挖坑，并在位置 B 上种树”就变得可行了。

这带来一个有趣的结论：我们（也许）(9)总是可以通过添加一层数据抽象，来将“逻辑的”时序依赖，变成“数据的”时序依赖。例如我们此前提到的：

□ 函数 B 用于计算函数 A 的执行时长，则函数 B 必须在函数 A 逻辑结束之后执行，

就可以理解为：

□ 设有时间戳 X，函数 B 修改该时间戳，而函数 A 统计该时间戳的修改。

两个逻辑作用于同一个数据（无论它们分别拿该数据来做何种处理），这是一个明显的特性。当这种特性存在时，我们称该数据是多个逻辑的数据依赖(10)。仅当我们能够通过对数据依赖的分析，而非对动态的逻辑执行结果的分析，就能确立这种依赖时，我们才可以自动化地分布与协调这些分布。

换言之，分布方式的确立以及在多个分布逻辑之间的协调等问题，都可以转化为对数据依赖的处理。再综合我们此前讨论的“数据的不可分布能够通过复杂的逻辑来解决”，最终可以得出这样一个结论：逻辑的不可分布并非无解，它最终可以被聚焦于“用于处理数据依赖问题的逻辑”的复杂性。

二

一个简单的数据依赖的模型可以是这样的(11)：

在这个模型中，若 foo_1()与 foo_2()是分布的，我们最终将不知道 x 如何分布，因此它们都必须使用数据 x 的全集。

如前所述，我们承认这种状况必然存在，无论它是发生于 foo()函数的循环展开，还是用于解决 foo_1，…，foo_N 之间逻辑的时序依赖问题。我们的问题仅仅在于：如何让 foo_1，…，foo_N 都持有这个数据 x。

我们必须再次确认“x 的全集”的含义。若“x 的全集”是一个有限的数据集合(12)，例如某个内存块或者某个对象，那么“持有这个数据 x”意味着对这个 x 的全集的任意操作。这样的全集应当包括三个信息：数据、数据的操作以及数据的状态。

其一，x 包括数据 x'。显然，x 应当包括 x'，它必然是可计算数据的一部分。尽管 x'在概念上能够指代可计算数据的全体，但在应用中仅是其部分，这就如同变量在概念上指代任何数

据，但在具体的代码中只表达确定数据。

其二，x 包括上述数据集合 x'的所有已知操作 x"。对于某个确定的时间（所谓“已知”，便是与时间相关的），x'具有确定的操作 x"。这基于两条理由：首先，使得 x'在某一时间的操作具有不确定性——也就是说，既能够应用某个操作，又不可应用该操作——的逻辑是不存在的(13)；其次，若存在在将来可能发生于 x'的其他操作，那么这些操作与 x'也将构成其他的数据集合。

其三，x 包括指示它自身的状态 Sx。Sx 是 x 之于时间的信息，这意味着 x 在一个持续过程中可以被分别处理，这个信息既可以看做是 x 之于时间的依赖，也可以看做是 x 之于“某时的处理者”的依赖。

综上，x 必须是{x'，x"，Sx}的全集。

三

在 x 这个集合中的 Sx 只是数据，而并不应当包括 Sx 的操作。这是考虑到 Sx 本身不能再有任何的依赖——包括时间，因此它必须是一个含义与可操作性都非常明确的状态值。若使 Sx 与其操作都存在于同一个集合中，则它们必须只能是一个可分布系统的“数据全集 x2”的一部分，并且该系统的数据集 x2 中也就存在相对应的。

作出这一点限制才可以使 x 作为一个独立概念加以考察，并且可以将“数据”与“数据的状态”在概念上有效地区分开来。由于任何状态

□ 本身也是数据

□ 也能被作为数据复合到一个“数据全集”中去

□ 也能存有其确定的操作

因此我们可以在程序中传递它、处理它，而不是视其为一个状态。

作为“状态”本身，Sx 在现在或将来，或是在“数据全集”的任意一个持有者手中，都必须以及必然是含义与可操作性(14)完全明确的。例如，它仅仅是一个“0、1”状态，它的操作仅有唯一的“逻辑非”操作。当然，状态可能更为复杂，但只要有与之相应的、明确的、完整的逻辑，并且这些逻辑可以独立于这一状态，可以由不同的持有者实现，我们都可以将这样的状态称为 Sx。

数据依赖在概念上只表明多个逻辑作用于同一个数据，它最终将被表达为面向{x'，x"，Sx}的操作，其中{x'，x"}表明被依赖的数据与其可确定的行为，而{Sx}表明一个有明确含义与可操作性的状态。

但我们并没有限制“多个逻辑”之间的时序关系。也就是说，数据依赖只用“状态”来表明：多个逻辑与数据的全集 x 都存在关系，但并不表明是一个并行系统下的关系，或是一个串行

系统下的关系。例如，“（对同一数据的）多读单写”显然是数据依赖的，但在多读时多个逻辑之间是并行的，而在单写时它们却是串行的。

在我们的定义中，所谓“并发多任务”系统，是面向数据依赖的一个实现。例如，上述“多读单写”系统是可以依赖针对于某个状态的一组逻辑来实现的。更确切地说，所谓“并发模型”，其实就是：

□　对上述状态定义一个操作集，并

□　在多个逻辑中实现该操作集。

第三节　消息

一

我们为什么需要状态？

事实上我们是把两种时序依赖都最终归结于“数据依赖”，进而将这种依赖关系委托于这个“（数据全集中的）状态”。并且，我们要求状态本身只是纯粹的数据，而将状态的相关逻辑视作公共规约。我们其实是在讨论如图 3-15 所示的模型：

图 3-15　将状态从纯数据中剥离，并讨论与逻辑步骤（Step）间的关系

这一模型的本质仍是基于“逻辑作用于数据”这一思想。也就是说，它是基于命令式语言设计范式：我们将状态{Sx}视为{x'，x"}的操作句柄，并通过某种规则在 Step1，…，StepN 之间传递该句柄的执有权（类似于一种令牌）。

既然 Sx 的抽象含义要求“含义与可操作性”明确，同时我们也知道，函数既包含一个数值含义的传出，也包含可操作性明确的逻辑，可见函数在抽象概念上是满足“状态”的两个要素的。那么，Sx 本身为什么不能是一个函数呢？

问题仅在于，“状态”作为数据含义时是与{x'，x"}相关的，而“函数”作为数据含义时是指它的传出。也就是说：

可以在数据上表明：

但这也导致我们无法通过 return 来表明 foo()这个函数的（数据含义的）状态(15)。

解决这一问题的方法，就是消息。例如：

这个例子中，foo()向 bar()发送了一个消息'…'，这个消息用于指明 foo 当前所持有的数据的某个状态。由于 foo()可以持有多个数据——多个数据全集，或者由多个子集构成的集合——所以它也可以发送一个复合信息的消息，或用某个单一消息来指示所有数据的状态。

二

我们看到，消息本身跟状态是同义的——具有明确的“含义与可操作性”，只是在函数（以及函数式语言设计范式）中，接收者可以视为发出者一定持有了{x'x"}，并通过消息 M 来通告了上述数据的状态(16)。

消息发布与处理，成为多个函数之间的一种关系，如图 3-16 所示。

图 3-16　消息的发布与处理：在整个逻辑步骤（Step）链条上的数据隔离

在这一关系中：

□　Step1 与 Step2 之间的时序关系与数据依赖关系是不明确的：所有类似的关系与依赖变成了它们之于消息 M 的处理规则，亦即是更复杂的逻辑(17)；

□　数据之间的关系被彻底屏蔽了：Step2 并不依赖 Step1 的任何数据，包括它的入口数据或出口数据(18)。

表面上看，消息类似于调用，例如我们可以将 Step1 与 Step2 之间的关系看做：

一方面，这种类似“调用关系”的形式的确意味着 Step2 在时序上依赖 Step1；但另一方面，在我们讨论的消息中，实际并没有“调用”这层关系。也就是说，Step1 向 Step2 发出消息 M 之后，就可以异步地执行自身的逻辑了，并不需要等待 Step2 的执行，也不依赖 Step2 的结果数据(19)。

消息并没有类似回调的性质，尽管消息可以实现这一性质。Step1 发出消息 M 这一行为，并不表明 Step2 必然执行某种由 Step1 决定的逻辑（例如回调函数），也并不表明消息 M 是某

个函数的数据依赖。

消息也没有类型与结构的约束。一个消息以何种复杂程度的方式来记录消息体，是 Step1 与 Step2 之间或者 Step2 与 StepN 之间的约定。除了约定的双方或多方对规则的强约束外，消息本身并不存在语言层面或计算机系统层面的约束。因此，基于消息的模型很容易适应复杂的计算环境。

三

状态与消息其实是一个统而一之的东西，它们不过是两个程序设计范式对同一事物的不同叙述。这样我们又回到了最初讨论问题的方式，即“两个侧像”。

回顾此前的讨论，我们曾经设定数据的全部性质中有一个不可或缺的子集为标识、值和确定性。其中值是数据的本义，标识是人与计算机进行沟通时所需的抽象定义(20)，确定性则是数据与使用这些数据的计算系统的最终极的问题(21)。基本上，我们可以将数据在计算机语言中抽象描述为图 3-17(22)：

图 3-17　数据在计算机语言中的种种抽象与概念递进关系

但是我们上面讨论的状态，是否需要全部三个性质呢？

这一设问的关键在于：我们将状态 Sx 作为数据全集 x 的一个分量，就意味着 Sx 的三种性质与 x 的其他分量 x'、x"必须是协同分布的(23)。因此设问若成立，就意味着任意一个数据在其分布过程中都要保证这三种特性的协同分布。例如数据 A，其分布为 A1，…，An，在同一时刻 A1，…，An 都必须是同时确定的，或同时不确定的。又例如——以其字面上的含义来看——我们必然面临类似的需求：在一个分布系统中的多个子系统中，要求同一个状态使用完全相同的标识。

对于状态，Peter Van Roy 是将它作为多种范式中对数据的不同理解来看待的(24)：

状态是记忆信息的一种能力，更精确地说，是及时存贮值序列的能力。它的表现能力极大地受到其所在范式的影响。我们将其表现能力划分出四个等级，它们的不同在于状态是未命名的或命名的、确定的或非确定的，以及串行的或并发的……非确定性对真实世界中的交互很重要（例如在“客户端/服务器”编程模型中），而命名状态则对模块化有着相当重要的意义。

在这一观念中，状态是一个可变的数据：表达某个存储位置上、某时的信息。换言之，数据的不确定性——包括其静态的确定性——表现为状态。这使得状态在概念上很类似于变量，不过这也并没有什么不妥，毕竟在大多数语言中，我们的确是用变量来实现状态以及相关的机制。基于此，Peter Van Roy 非常深入地剖析了状态的数据性质。

把 Peter Van Roy 的思考逆向探讨一番，我们可以得出这样一个结论：当不考虑一个存储位置上的命名特性时，它就既非变量或常量，也非某个确定的运算对象（例如“对象”等高级

的抽象概念），而只是一种更加泛义的“状态”；同时存储这个状态本身的事物，由于没有位置、时序等概念，所以借用我们之前使用过的名词，可以称之为“cells”。

一旦我们不考虑这个存储位置（cells）本身的物理限制，而只考虑其中数据本身的、值的含义时，我们也就可以脱离系统（例如某个系统的多个子系统）地称之为“消息”。消息既是系统之间的约定，也可以视为是游离于系统之间的、借着这些约定来约束系统的数据。

综上所述，状态与消息的抽象概念的区别在于：前者往往带有位置之于存储，或数据之于逻辑的含义；后者则将位置与逻辑含义都抽取掉，只认为消息是一个约定所需的数据部分。图 3-18 表达了二者的关系(25)。

图 3-18　约定规格：值性质下（状态与消息）与变量性质下（结构与对象）的抽象概念对比

其中，对于消息：

□　其数据规格部分是应用中的所需，而非概念上的必需；

□　约定所需的逻辑部分是由其他系统根据约定实现的；

□　约定（在概念上）是可遵守、可更新，以及可违约的。

既然其约定是一个非数据的规约，是系统间在逻辑实现上的约束，并且事实上这一约束也可能不被遵守，所以可以说：

消息就是消息，消息没有任何特定含义的性质。

第四节　系统

一

煮鸡蛋，大概是最简单的美食了。这个世界上已知的最完美而又同时是最简单的煮鸡蛋方法就是：把鸡蛋和水放在锅里，煮。

有两个极简单但不强制的限定条件：其一，水要漫过鸡蛋；其二，烧火的时间要够把鸡蛋煮熟。所谓“不强制”，是指在大多数情况下：水不漫过鸡蛋也是可以的，只要别煮干；而鸡蛋煮得不是很熟，或者多煮了一些时间也没关系。

而且，煮一锅鸡蛋与煮一个鸡蛋的方法和限制条件居然是完全一样的！这意味着，我们煮鸡蛋的算法以及作为数据的鸡蛋，乃至煮鸡蛋所需的整个系统，天生就是支持完美分布的！

但问题是，你并不能确保做好这份美食。

二

当一口锅架在那里的时候，你没有办法保证：

□ 没有人或其他动物突然闯入，打翻它；

□ 没有地震、飓风等自然灾害发生，破坏它；

□ 没有柴尽水干的极限情况；

□ 时间不会停止；

□ 空气不会凝固；

□ 战争不会爆发；

□ ……

事实上，除开这些宏观因素，你也不能保证这样一些眼下所需的条件：

□ 没有不好的鸡蛋；

□ 没有不爆的木柴；

□ 没有不漏的锅；

□ 没有不塌的灶；

□ 端锅时不烫到手指；

□ 取蛋时不滚到地上；

□ ……

更有甚者，你想煮熟鸡蛋，你却根本不知道：

□ 如何看出这颗鸡蛋已经熟了；

□ 如何保证一锅鸡蛋的每一个都熟了；

□ 保证每一个享用美食的人都吃到他们认为“熟了”的鸡蛋；

□ ……

所以，你看，我当年打算学厨的时候，就被我父亲断然拒绝了。

三

Joy 都可以煮熟鸡蛋(26)，但作为架构师的我，却只得出了一个结论：煮熟鸡蛋的可能性似乎连 1%都不到。

为什么？

因为当架构师看待一件事物时，他看到的是它在系统中的全局问题；而一个实施者，却只需要去做局部的那个部分。当我们把“全局”这个问题牵扯进来之后，任何事情都变得有无限种可能性，因为任何一个系统都是更大系统的局部，而且这个命题是无限递归的。

蝴蝶的翅膀，只是一个生物学的问题；蝴蝶翅膀的扇动，只是一个空气动力学的问题；如此等等。但如果我们不能像这样地限定一个系统的领域边界，那么架构师所能得出的结论永远就只有一个：我们什么也做不出来。

所以在大型系统（以及任何一个需要架构的系统）中，我们得承认两点事实：其一，我们的系统永远都只是更大系统的局部；其二，我们只能在实施中关注可控领域中的可控因素。

那么，如何应付未知系统以及那些非可控因素呢？

四

答案是：忽略。

忽略掉一部分问题，是我们在系统实施中所必需的策略。(27)除了能降低当前系统与外部系统的耦合关系、减少系统处理异常情况时的成本之外，更重要的一点是，这样的策略有助于我们快速地将问题聚焦到该系统的核心领域。

除开计算机的两个关键问题——计算总量与数据总量之外，我们将领域之于系统的主要需求变成一个简单的定义：计算系统如何应对外部系统的需要。而进一步地考察这一需要，就可以发现：在“可计算”方面，任意领域的系统对“当前系统”的需要只有两个，即寻求计算资源，或寻求数据资源。

换言之，任何一个子系统——作为一个更大系统的组成部件——对外部系统只需要承担这样两种责任(28)：

□ 维护可对外公示的状态，使得外部系统可以将明确的行为（逻辑）施于自身；

□ 接受外部的消息，使得外部可以通过传递数据来影响自身。

对于接下来要讨论的问题来说，“系统由子系统构成”是一个基本设定，并且我们还将基于这一点来讨论通讯与验证的问题。子系统、通信与验证作为系统的三个基础部件，是我们在“第四编　架构的思想与指导原则”中讨论架构组成论的一个基础，而本编的下一章将基于这些部件，来讨论与架构实施相关的技术方法。

————————————————————————

(1) 这其实并不那么明显。其一，处理（process）是函数的基本抽象含义，如同它在数学中的含义是求解；其二，数据（data）是处理的具体对象，这是函数入口参数的基本抽象含义，如同数学中的公式是函数，而代入公式的那些数才是求解的问题（即数据）。

(2) 引导光盘、引导软盘，以及硬盘活动分区的引导扇区等。

(3) 这里仅基于 Windows 系统的“进程－线程”模型进行讨论。事实上这些逻辑单元在不同系统中有着多种类型、多种层次与关系的抽象，例如作业（Job）、任务（Task）、进程（Process）、线程（Thread）、流（Flow）、事务（Work）、会话（Session）等。

(4) 这句话的意思是说：我们只好让那些不能拆分的逻辑运行在同一个处理单元中。

(5) 如果使用一个自动分布程序来处理函数 foo()，我们显然只需要进行完整的语法扫描，以确定 foo_1 与 foo_2 各自使用的那一部分数据即可。

(6) 对于在分布环境下的“函数的参数界面”，在 Erlang 中可以理解为消息，而在另外一些基于数据库、数据中心或数据结点的解决方案中，可以理解为持锁的数据项或结点。

(7) 在这个问题中，A1，…，An 是否是相同的逻辑呢？这并非一个关键问题，因为我们现在仅在讨论数据的分布。

(8) 这也提出了数据规划的必要性问题：若我们现在有机会规划这些数据，那么必然能降低将来处理它们的成本。

(9) 很遗憾我无法证实这一点，但就目前的实践来说，这总是成立的。

(10) 我们已经提到多个与“依赖”相关的概念。若我们只是单独使用“依赖”，则表明它是自然语言中的、表明多个事或物之间存有依存关系的含义。当我们使用“时序依赖”时，它是包括逻辑的与数据的时序依赖两种情况。当我们使用“数据依赖”时，我们是用这个概念来统一了上述两种时序依赖。

(11) 这个示例基于 Peter Van Roy 和 Seif Hardi 在《计算机编程的概念、技术和模型》（MIT Press，2004）一书中对“可观测的非确定性”（Observable nondeterminism）的讨论，原示例使用 Oz 语言。

(12) 若“x 的全集”是一个无限的、增量的数据集合，那么我们这里的规则就未必适用了。对于这种情况，通常有两种处理方案：其一，若该增量具有唯一的起始或标识，则可以以之作为数据全集的映射，并进一步将它映射其后的状态；其二，若该增量是持续发生的，则它应当被抽象为一个与时间相关的函数，因而任意逻辑在任意时刻都只可能持有该函数运行结果的一个片断——换言之，它是不可能被持有全集的。

在对“x 的全集”这一概念的实现上，若语言中存在变量，则变量本身即可被理解为上述的“具有唯一的起始或标识”的无限增量数据，这是它适用于上述讨论的根本原因。若语言中不存在变量，则每一个数据——在它被唯一持有时——就是全集。其三，闭包这一概念的提出，在于无论数据在时间线上映像为何，在任意时刻闭包总代表着数据全集。

(13) 此前我们已经讨论过“计算的不确定性是对机器计算是否有价值的终极拷问”这一问题。它既是我们能通过“计算机语言”进行机器计算的基础，也是使得我们能够正确地将这样的计算应用于一个“数据全集”的基础。

(14) 我们稍后会再来讨论这一限制的实际含义。

(15) return 既已用于表明 foo 的数据含义，则必然无法用于表示“含义与可操作性确定的、foo()的整体含义”。即它不能既表示 foo()的部分含义，同时又表示 foo()的整体含义。这种“既是又非”有违于基本逻辑的矛盾律。

(16) 我们用到了“通告”这个词，意味着消息的发布可以是单个的，也可以是多个的。前一种情况可以实现令牌式的同步，后一种则可以实现多读单写式的同步。

(17) 这些逻辑基于 M 的数据含义，即{m'，m"}，它被 Step2 作为私有的数据信息加以处理。

(18) 同样地，当 Step2 发出消息 N 时，N 与 M，或 N 与{m'，m"}并没有必然关系。

(19) 这意味着，“发出消息”对于 Step1 来说是一个“有确定结果的确定行为”。这一确定结果是“没结果”，因为在 Step1 中不存在对这一结果任何有意义的依赖。而“函数调用”正好是存有对某种结果的依赖的。

(20) 这是程序设计语言的需求，即标识符（token）、名称（name）或词（word/keyword）的本义。

(21) 在之前的种种讨论（事实上也包括接下来的讨论）中，凡涉及对“数据性质中的确定性”的追问，必然会对我们在计算模型、语言范式等方面提出新的挑战。

(22) 在从变量到常量的概念变迁中，某些语言并不引入“常量”这一概念，而是通过强调“变量不可写”来强制数据必须具有确定性。在类似的语言范式（例如数据流范式）的基础

观念中，通常还涉及变量仅作为标识符而未有确定值时的抽象。例如在 Oz 与 MapReduce 中，如果一个变量尚未绑定值，则它将阻止运算它的函数与任务（job 或 process），因为这时的运算也是不确定的。直到该变量存在一个确定值，阻止才得以解除。

(23) 这里的协同分布是指：未必一致，但至少是通过某种规则来约束。

(24) 引自 Peter Van Roy 对“主要编程范式”一图的说明。参见：www.info.ucl.ac.be/～pvr/paradigms.html

(25) 该图也从侧面表达了变量与状态在数据性质上的相似性，以及在结构化程序设计和面向对象程序设计中“可以以结构、对象等类型化数据来实现状态”这一事实。

(26) 其一，看过前言的人应该知道 Joy 是谁；其二，Joy 总是把鸡蛋煮得很熟，这是一种过度的安全措施。

(27) 对于灾害，一个简单而实用的策略是：备灾甚于防灾。例如考虑到海底光缆断裂，那么就让 30%的数据通过陆路光缆；考虑到地震频发，那么就将应用分布到不同地区的机房，并在各机房做全镜像数据。这样的措施也被扩展到类似的系统问题上，例如考虑到战争，就将重要的计算资源部署在中立国；考虑到断电，就要增加备用电源，以及设计物理解决方案。但总的来看，这一类措施都是“忽略问题本身”的。

(28) 或之一，或全部，取决于系统设计的具体策略。

第九章

系统的基本组织方法与原理

甲邀请乙：“周六我等你一起去打球”。这个邀请，很有可能会以甲无休止的等待而告终。无论乙是出于何种原因爽约，例如赴约途中被掉下的卫星碎片砸中，甲终将因为发起“一份失败的邀请”而懊恼一周、两周……从此他可能不再对乙抱以信任，进而对所有人都不再信任，于是他的“人际交往系统”受到种种限制，乃至出现失效，最终不得不求助于心理辅导……

这里我们看到了两出悲剧：一种是处理异常，例如乙的爽约；另一种是框架失效，例如甲的邀请。

这两种悲剧都会出现在我们的软件系统之中。因此我们的系统不稳定，既来自于客观上不可避免的异常，也来自于主观上框架设计的失败。

第一节　行为的组织及其抽象

一

在系统的定义及其规模化这两个方面，一个大型的系统与煮一个鸡蛋的系统并没有本质的区别；当这样的系统在数量指标上增长时，它面对的问题与煮一锅鸡蛋也是相类似的。但我们并不是要试图去讨论“它是不是一个系统”，或者“它是不是一个在数据量或运算量上足够‘大’的系统”，因为总的来说，这两个方面并不是我们在这本书中定义“系统”（system）这个开发规模时的本义。

反观“应用”（application）这个规模，它关注特定的应用领域，却并不强调对该领域之上的整个行业链条的观察。例如开发一个看图软件，我们并不会将数码打印行业的功能给整合进去。又例如设计一个资产管理软件（有人也称为“某某系统”），我们并不会将相应公司的人员管理体系也纳入设计范畴。特定应用领域决定了应用的特性及其在领域知识上的复杂程度，并且在一定程度上，也表明其产品构件可以在相同领域中复用。

而“系统”这个规模是包含“应用”的，因为这一规模的定义本身就是由跨领域引申而来。例如在线支付系统，它本身涉及 Web 应用、支付应用、金融类应用、在线交易类应用，同时也涉及服务器端数据安全和挖掘、大规模数据和计算的分布以及容灾等领域。不同的领域有着各自的领域知识，以及相对独立的开发技术、框架与业务模型。

但系统本身的复杂性并不是由这些领域带来的，正如我曾经说过：牌局的复杂性其实不是由“分开牌”这个动作导致的，而是由“交叉牌”这个动作导致的。我们将一个系统规划为各个领域，这些领域自身的问题仍然只在“应用”这个规模级别之上，而领域之间的交叉、交互才是“系统”这个规模自身的问题。

这也是“系统何以作为一个规模”这样的问题的根源。

二

当然，数据量或运算量上的规模依然是系统的一个（外在的、表现上的、实际应用中的）问题。就目前的实践而言，我们事实上也把它们独立为计算机系统开发中的领域，例如分布式计算领域以及分布式存储领域(1)。

分布并不等于并发，这一点我们此前已经讨论过。但分布必然意味着要处理逻辑与数据之间的耦合关系：其一，耦合紧密的逻辑不易拆分，也就不易分布；其二，逻辑与数据紧密耦合，也存有类似的问题。现实中的通用应用开发语言（例如 C、Java 等）是在“算法+数据结构=程序”这样的结构化理论上发展的，为了增强语言的通用性，其结构化的特性非常明显。这并不利于数据与逻辑的解耦，例如在一个既存的面向对象应用中，想要将对象的数据成员与

方法成员解耦并分布，是一件不可想象的事情。但在这个问题上，并不是说具体的语言有多大的问题，而是这个程序设计范式本身的理念与我们讨论的背景并不适配。

在现在的大型通用应用开发语言(2)中，接口的引入是解决上述问题的一个良好实践。但接口带来的是面向行为的设计理念，例如在这样的环境下，我们讨论的问题通常是：某个对象或某个子系统所具有的哪些行为，是可以被抽象为接口的。而实践过程中的这些具体工作，与对象本身的设计以及与面向对象系统的设计是没有多少关系的。也就是说，接口设计与面向对象设计是两回事，只是实践中把两者综合在一起，并试图解决后者的一些实际问题。

这些“实际问题”，具体来说，就是“逻辑如何分布”(3)。

另一方面，现实的软件开发中的“数据”，并不总是和对象一样地通过一个实体（instance）与方法绑定起来。我们面临的这些数据(4)要么是结构化的，例如结构化文本或结构化数据库，要么是非结构化的，例如待索引的文件或者流式数据。这意味着，通用应用开发语言对于处理大数据量几乎毫无帮助。例如在实际工作中，如果这些数据存在于关系型数据库中，我们将使用 SQL 语言来操作；如果它们存在于 NoSQL（非关系型）数据库中，我们就会采用类似于数据流式语言或其他特定的数据处理语言来操作。

所以综合来说，即使仅仅观察“数据+算法”这样的软件开发本质工作，大型系统也被具体分成多个领域了。而软件开发进入领域细分时代的大背景，才是像“云计算”这样的理念得以提出的基础。

换言之，我们一再讨论的分布问题，本身也可以被理解为领域问题。

三

系统由领域组成，领域又由细分领域组成……由此我们最终所讨论的、具体的、作为组成部件的领域被表达为：

□ 包含了特定领域知识，

□ 提供了基于上述知识的处理能力，

□ 反馈了在领域间可理解信息的一组行为。

对于这样一组行为，我们称之为“（该子系统/领域的）接口”。请注意，我们讨论系统中的接口时，它是脱离具体语言的。我们需要这样一层抽象概念的根本原因在于：

（1）将领域问题转义为一组逻辑的界面；

（2）将领域间有无关系，转义为“领域对领域（调用者与被调用者）”的接口是否可达；

（3）屏蔽领域内的数据性质(5)，迫使开发者必须通过特定的设计来解决领域间的数据问题（例如数据类型、数据依赖等）。

在这三种原因或可称为三类需求及其解决手段中，第一种和第二种是显性的，它们由接口的抽象特性决定：其一，表达为一组方法；其二，能否使用这些方法，取决于能否获得接口。

第三种则是隐性的，并且也是决定系统稳定性的最主要因素。事实上在这样的背景之下，开发者能够选择的方案也相对有限，主要包括远程方法或远程对象、消息/状态、序列化、分布等。

最终，我们将提供一组接口的系统组成构件称为一个“服务”，或一个“结点”。前者是功能视角下的一个概念，后者则基于部署视角。(6)但总的来说，服务/结点都意味“非本地”的支持，这也可以理解为对“跨领域的需求”的支持。

第二节　领域间的组织

一

系统与子系统之间面临的第一个问题，是系统的整体部署。系统的部署方案几乎决定或限制了大多数有关系统的决策，其中首当其冲的是数据的结构化与预结构化问题。

数据在哪里？数据的型式是什么？数据的量级、频度与粒度如何？这是系统整体部署中避无可避的三个关键问题。我们先讨论 B/S 架构下的数据，通常它们的大多数数据项表现为微数据，其数据粒度小、频度高、可靠性差。

频度高意味着单次处理数据的 CPU 占用要尽可能小，这是维持大的系统处理能力的诀窍。但如果一笔数据的结构不确定，发现数据有效性以及决策计算路径的代价就将变得极其巨大。举例来说，Java 的 Web 应用框架大多提供“将 HTTP Request 转换为一个数据对象”的能力，其优势是在应用层中不需要关心数据来源与有效性。这一方面可以让应用层用类同的方式处理请求，另一方面也可以屏蔽一些框架逻辑，例如通过注解（annotation）来做数据验证。而且在第三个方面，这还可以将服务端与请求端无缝隔离，例如一个应用处理逻辑并不关心请求是来自于 Web 客户端，还是来自于 Open API 接口。

但是“将 HTTP Request 转换为一个数据对象”的代价极其巨大，其本质上是在服务端、在应用服务器的框架层上进行数据的预结构化。如果我们理解这一点，那么我们在 Web 客户端对数据格式进行预处理，并与服务器端将这一规格协商作为协议，其效果将是十分显著的。例如大多数 B/S 应用会面临登录验证的问题，它们需要处理大抵三类需求：

□　如果用户未登录，则一些客户端请求无效；

□　如果用户未登录，则一些客户端请求被保持，并在再次登录后提供服务；

□ 如果用户已登录，则可以为客户端请求提供正常服务。

“登录与否”有许多种特定的验证方法，例如是否存在用户名（UserName），是否存在用户会话（SessionId），是否存在验证数据（CertData）等。但我们这里先关注“验证数据的获取”，这在一个 HTTP Request 中有几种来源(7)：Cookie、Get Fields、Post URL 和 Post Data。其中 Get Fields 与 Post URL 是同类型的东西，因为“Method 为 Get 的表单”中的字段最终会作为 URL 请求中的字段信息提交到服务端，URL 中包括 Path、Action 和 Fields 等。

那么请问，验证信息（以 SessionId 为例）应该在哪儿呢？

二

一个 HTTP Request 通常要经过与服务器端的多次通信才能完成提交，过程大致如下：

□ Web 服务器只负责接收数据，并在适当的时候将执行权转换到应用的框架层（Framework）；

□ 框架层会解析这些接收到的数据，构建（Build）请求对象或将某些数据持久化（例如文件上传）；

□ 最后，框架层将一个请求对象作为应用可以理解的数据格式，例如 Java 对象，提供给应用逻辑来处理。

在这个过程中，如果数据有效性验证发生在应用逻辑中，将是效率最差的。相较而言，较好的位置是放在框架层“构建（Build）请求对象”这一环节中（A 方案），而更好的位置则是放在服务器接收数据这一环节中（B 方案）。

B 方案通常是通过 Web 服务器插件来实现，它不太方便的地方在于：如果在其中加入大量逻辑，对服务器整体的稳定性将构成影响，并且这些逻辑可能是重复而无意义的。举例来说，如果我们试图在 URL 中预结构化请求数据，并在 B 方案的插件中来解决，那么 Web 服务器需要解析 HTTP 请求的 Head 部分，读取 URL 并解码、定位字段、验证……而这些过程既拖慢了服务器响应，而且相同的逻辑还需要在框架层上再做一次。所以虽然 B 方案中看起来是“更好的位置”，但对于我们这里要解决的问题来说，却是相当不妥的。(8)

A 方案通常是应用服务器内部的逻辑。我们只提及一个问题：需要“将 HTTP Request 转换为一个数据对象”完全完成之后，才能进行后续处理吗？这个问题是“数据如何进行结构化，以及在哪里进行结构化”的核心。举例来说，如果客户端具备规格化数据的能力，那么我们可以要求“在一个有效的 HTTP Request 中，字段数据的第一项必须是 SessionId”(9)。将这作为一个约定时，我们看看 B/S 框架的各层之间是否能实现这一协议：

□ 在浏览器端，可以通过在 URL 后添加字段数据的方法使 Get/Post 请求都满足上一协议。以 Post 请求为例，当请求的 URL 中带有“?SessionId=xxxx”时，它可以被提交到服务端作为 Request Fields 数据。

□ 在服务端，当解析 HTTP Request 并尝试 Build 时，可以从 URL 中顺序读取到 Action 信息，以及接下来的 SessionId 信息，这应当是在一次解析过程的前后连续动作中发生的（先是 Action，接下来是 Query Fileds）。因此可以按顺序逻辑来实现“判断第一个 Field 是否是 SessionId，并判断它是否有效”这一行为。

□ 在服务端，如果上述判断为“否”，则框架层完全有能力响应浏览器端请求，例如（1）重定向到登录页面，或（2）返回标准错误信息，或（3）保存当前请求作为事务数据，以开始一个“登录+重新提交”的过程。

接下来让我们看看这一系列的复杂过程有何收益？其一，当一个客户端请求发生时，服务器可以在没有完全接收数据的情况下作出响应，这提高了客户端的体验；其二，一个无效的请求在抵达（执行过程将更占用 CPU 的）应用层之前，在仅需最少的执行周期的情况下，就已经获得了正确的判断信息并处理了。

从这一过程中，我们可以得到一个结论：数据的结构化阶段离处理阶段越远，其系统的整体收益也就越高，即数据要“尽可能早地”结构化。但结构化是一个逻辑过程，需要相应的部署环节的支持，例如我们需要考虑 Web 客户端层的整体系统结构。

只有在部署许可的情况下，我们才有讨论“结构化与预结构化”的位置的可能性。例如基于客户端与基于 Web 的即时信息（IM，Instant Messaging）软件，就非常不同，前者对数据的预结构化能力非常有限。通常在系统中，我们寄期望于数据的发出者具备结构化的能力，并尽量提供在多个子系统之间的标准（Standard）、基础（Base）和基元（Meta）信息格式。如果这一点不能满足，我们也通常不在具体应用逻辑中处理格式，而是将这一过程交由中间层来实现。例如将来源数据读取为内存数据库，或添加数据接收服务器，来将端口或远端数据转换为系统内部支持的格式。

以尽量远离处理逻辑的方式实现数据规格化，即使仅仅是形成了类似“Name/Value”这样粗粒度的数据索引(10)，也会给后续处理逻辑带来巨大的收益。

三

一般性的数据处理的思路，是通过多次的数据筛选将待处理数据层层精化；在得到所需的处理数据之后，将这些处理数据从数据层迁移到应用层；最后，由应用层来处理之。关系型数据的基本原则，是根据“应用需要”来构建库中数据项次之间的关系，并规格化存储（包括索引与键等）。这一原则其实是有利于实现上述的“一般性思路”。例如我们通常通过在数据库中执行 SQL 语句来得到一个数据子集，然后通过应用程序中的数据库客户端得到它，并转换为应用程序识别的数据对象再加以处理。

不过，这个思路有一个致命的问题：如果“应用逻辑所需要的处理数据”本身就是一个极大的数据量呢？例如搜索，无论用户输入怎样简略的一个关键字，他的原始意图都得是在整个数据全集中检索之。事实上，搜索的特殊性就在于任何一个应用逻辑都是面向数据全集的。关系型数据库对这一问题的解释是，检索行为应当交给数据库端来实现，进而相类似的、基于数据全集的应用逻辑也都应当交给数据库端来实现。

于是我们看到了基于关系型数据的数据处理得以进化的动力：由于数据从数据层迁移到应用层的成本过高（这包括数据本身的传输成本，以及在应用层实现相关处理逻辑的成本），因此将相关处理作为独立技术在数据层实现是必然之选。在这一层次上的需求已经不是传统的基于数据库管理（DBMS）技术所能应付的，例如联机事务处理（OLTP）试图在一个事务中完成对数据的切片、加锁与运算过程，而这往往导致整个应用层需要花大量时间来“等待”数据层的运算结果。

反思这个运算瓶颈的核心，关键在于关系型数据库对数据规划的理解，它本质上是强调数据项（Rows）间存在序列关系，以及数据列（Cols）间存在的键关系。这两类关系使得大数据处理时，无论基于列的纵向拆分或是基于行的横向拆分都面临“如何处理关系”的问题。例如将一个类似于日志数据的大表（Rows 项次过大）作横向拆分，我们必须在处理层——无论是数据库的事务或是应用层的逻辑代码——中解决对“基于年、月、日或季度、周等时间区段”的数据的归并以及跨区段处理的问题。又例如将一个类似于订单数据的宽表（Cols 项次过大）作纵向拆分，我们就必须通过多表的 Join 来得到最终所需的运算数据，这又涉及 Keys 的维护，从而加大了应用端或数据库端的逻辑复杂程度。在这个问题上，联机分析处理（OLAP）在本质上也只是通过多次的、渐增的数据规划来强化数据关系，通过需求规划来增加中间数据，从而减少单次处理的时间。

那么是否需要将这两类关系“解锁”？去除掉这两类关系的数据，又是何种数据呢？

四

对这个问题的思考将数据处理带入了 NoSQL 数据库的领域。其中具有一定代表性的是 Key-Value 型的非关系型数据存储方案，即基于 Keys 的分布来存储数据项，这使得数据项在 Rows 维度上是无关系的。对于类似日志这样“天然存在 Rows 关系”的数据，Key-Value 的解决方案是分段/区存储，这其实是在规划阶段对数据进行了横向拆分，而这事实上又导致了小规模处理此类数据时的不便。

Key-Value 数据很好地从纵向规划的角度将大数据切分成多个数据簇。在这个过程中，Key-Value 是允许存在冗余开销的。对于关系型数据库来说，这类似于索引的存储开销。但正因为一笔数据在多个位置冗余，所以 Key-Value 面临同一数据的多点更新问题，这使得数据的 Update 周期变长。这一策略换取的收益是：通过一个单一入口获取信息变得迅速而简洁，并且可以用较小代价来应付多层 Key-Value 关系，而后者正是关系型数据库的弱项。

如果 SQL 是被视为一门语言（事实上它既是语言，也是一类数据库系统的称谓），那么 NoSQL 就是另一类与之本质上存有不同的语言的统称。如此一来，不得不观察 SQL 的语言特性。简要地说：SQL 的语言特性就是基于批量数据的提取与后续操作，它在定义上总是基于“一个数据子集”来思考的。与之相对照，NoSQL 却是基于“如果数据子集的获取是一个过程，那么这个过程本身是否可以被分布”来思考。

在 NoSQL 的思维方式中，数据是通过一大批获取过程、在一系列被分布的数据中“收集”而最终得到的一个结果集。在这一过程中，将“数据获取”分成多个逻辑以及多个匹配路径是必需的。NoSQL 的这一设问，正好与我们此前讨论的“是数据的拆分，还是逻辑的拆分”

不谋而合。在 NoSQL 环境下，大数据将不再是一个中心，而是一个被提前规划、提取与规格化的“数据广场”。一个很好的比喻是这样的：关系型数据库将数据理解为矩阵化的养鸡场，而 NoSQL 环境下面对的数据环境如同鸽子广场。要在这样的数据环境中实现运算，“从不同维度来规划、标识与筛选鸽子”将是数据处理者的责任，而不再是数据库厂商的“不传之秘”。

这些（既是负担，也是自由的）责任在 NoSQL 方案下仍然是通过语言来实现的，例如 MapReduce（如果我们把它看成是一种编程范式的话）。在这一模型下，Map 过程用于从一个数据系列中获得数据子集，而 Reduce 过程则将些子集再次规格化为新数据：如果还需要后续处理，则持续进行 Map/Reduce 以得到适合应用层处理的数据。这一设计思想是基于（比结构化编程范式）更为原始的数据流编程范式（Data flow programming paradigm），从本质上来说，这也是函数式的思维模式。这些对程序设计语言范式的考察揭示了大数据处理下的新思维体系与传统体系的差异。将 NoSQL 与 SQL 对比来看：

□ NoSQL 代表对“数据可变”的理解，而 SQL 代表对“逻辑可变”的理解；

□ NoSQL 认为逻辑（例如 MapReduce 的数据规格化过程）是不变的，应当让数据“流过”逻辑，进而得到数据某些方面的映像；而 SQL 认为数据（例如关系型数据的既定结构）是不可变的，应当让逻辑“作用于”数据，进而得到一个可呈现的观察。

五

数据或逻辑的不变性是这两类大型系统（以及系统开发语言）的核心区别，如图 3-19 所示的两种“PD 模型”(11)：

图 3-19　两种“PD 模型”：数据或逻辑的不变性

函数式语言，例如 Erlang 和 MapReduce，主张第一种系统模型（另一种模型在后文中另作讨论），因此适宜于编写逻辑确定的系统。它使数据 D 穿过逻辑，形成 D'，这一过程是确定的；而由 D'与其他的、后续的逻辑构成的部分（子系统或领域），并不影响当前系统的确定性。

但是我们也可能会注意到，在网络上产生数据（例如发表博客或提交回复）时，数据是动态产生的；而当这些数据被收集起来之后，我们展示它们时数据却是静态的。简单地说，数据收集和规格化，与基于数据的应用程序开发，看起来并不是一回事。在后者，即对于我们一般意义上的应用程序开发（而非数据处理），我们通常还是会选择第二种模型——让逻辑作用于数据。

而这一模型事实上并不简单。例如，即使我们的数据是确定的，但如果它本身是海量的、分布的、复杂结构的，又应当如何处理呢？仍以我们在上一小节讨论过“搜索”为例，如果一个搜索引擎已经获得了数以亿计的网页，分布在不同物理位置的存储设备中，又如何能让一个关键字搜索快速得到响应呢？

首先，真正发生一个“按 Key 搜索”的行为时，事实上是不会到具体网页中去“逐字节查找”的。关于“Search Keys”与“Page Keys”的关系以及“Page Keys”与存储的部署关系，将涉及非常多的系统策略，在此我们不细讨论。我们这里只关注“如何发生查找逻辑”这一个问题。而这个问题的本质是：如果逻辑所需处理的数据是不可迁移的，那么逻辑可否迁移？例如，如果我们的“按 Key 搜索”行为可以被分拆为多个子处理，那么能否将这些子处理“送入到”数据所在的集群（cluster，或物理服务器），并完成运算呢？

传递逻辑而不是传输数据，是海量数据运算的另一个思路，而这一思路依赖的条件是：逻辑的子过程的分拆是可能的、可控的(12)。在类似 MapReduce 的方案中，Map/Reduce Jobs 的执行就具有类似的特点。也就是说，我们必须关注另一个事实：

语言选型与系统架构在“数据与逻辑的可变性”上是可以互为补充的。

六

接下来我们讨论三种常见的、以数据为中心的服务：会话、登录与镜像。

会话这一服务，是将数据理解为公共数据与状态。通过会话来形成事务是一种常见的策略，但其可靠性是受置疑的。例如我们可以将用户在 Web 上的一次交易，看做是一个有着松散的事务关系的过程，从选择商品、下单、确认、支付以及转入到发货环节。这在整体上有着工作流的特点，在部分结点上有着事务性的要求。在这样的背景下，我们要求从选择商品开始建立一个会话，并在其后的行为中基于会话来实现处理逻辑。

将会话作为一个“状态”其实是非常危险的，因为 Web 交易中不可避免地会发生用户刷新页面导致的重复提交。因此，在同一个状态上出现两个等待逻辑的状况就会出现，若这种逻辑又正好是在事务型的结点上，例如支付，那么该怎么办呢？

在稍小一些的系统中，由于相关的领域和应用结点不是非常多，因而我们还有能力应付交叉逻辑中的种种锁关系。但如果系统的规模相当大，应用逻辑相当复杂，那我们就必须回到最原初的设定上进行重新思考：会话，是否应当将数据作为状态？

在此前我们讲到过，一个能保证确定性的“数据全集 x”必须是{x'，x"，Sx}整个集合，其中的 Sx 就是状态。将会话数据作为状态是典型的{x'，Sx}集合，因此它表现为数据确定而逻辑（x"）可变。我们之所以会在逻辑中出现锁关系，便是因为逻辑对 Sx 产生了依赖（而非对 x'产生依赖）。因而将“状态（Sx）”这一性质从“会话数据”中抽离是我们必然面临的问题。

一个可选的策略是加入消息服务。这基本上借鉴自语言设计中对并发的处理，例如在 Pascal/Delphi 系列中采用的共享状态模型，以及在 Erlang 中采用的消息传送模型。图 3-20 说明相关语言特性的演化(13)：

图 3-20　两种可选可替代的并发模型：共享状态模型与消息传递模型

这说明共享状态模型与消息传送模型是可以相互替代的。具体来说，如果我们试图从会话中抽取掉状态，则应该考虑在多个任务（task）中“有效地传递或限制传递”这一状态（Sx），而不是将 Sx 放到另一个数据结点中去。后面这种策略，以及之前提到的会话服务事实上都是因为 Sx 的存在而形成了数据单点。

当我们加入消息服务 M 之后（消息本身的“数据性质”中并不包括 Sx，并且也不包括逻辑 x"），任务 T1，…，Tn 之间便只因为“共享了会话数据”而变成相同的 n 个任务。当它们的“修改状态”请求被投送给 M 之后，M 的端口（或其他消息限制策略）将请求的消息信息队列化，进而将状态 Sx 中的时序信息拿掉(14)，因而在具体的、在消息服务 M 中的、有关消息响应的逻辑中，就不必再处理时序信息了。

在消息服务 M 中是否使用同一个会话数据作为上下文，是消息模型中的一个可选策略。这可以表达为对“（确定数据下的）PD 模型”的两种不同理解，如图 3-21 所示。

图 3-21　对确定数据下的 PD 模型的两种不同理解

在解释 1 中，认为逻辑 Pn 通过确定数据之后，得到的是一个包括数据映像的新逻辑 Pn'。在这种情况下，新逻辑是可以基于数据映像进一步求值的。这一思路可以理解为缓存，即在一个存储上加上 IO 逻辑。当通过 IO 逻辑得到数据时，我们并不关注数据是原始的确定数据，还是这一数据的一个映像，并且事实上我们也不关心映像与确定数据之间的同步问题，这是 IO 逻辑之下的服务层的责任。解释 2 则认为逻辑通过确定数据之后，得到的是可以用作后续计算的数据。因此，这一思路可以理解为函数连续调用，也是函数式语言的一个基本范式。

这两种模型都可以解释“消息服务 M 中是否使用同一个会话数据作为上下文”(15)：对于解释 1，要求会话数据是通过一个存取界面得到的；对于解释 2，要求会话数据是在消息 M 的处理逻辑中自行持有的（例如可以队列化、加锁，或使用类似闭包的方式得到一个映像）。

七

登录，是第二种以数据为中心的服务，它本质上就是以数据为单一结点的。也就是说，登录在逻辑概念上，就是数据单点的。登录的问题通常包括并发数量巨大、容易形成访问峰值，以及逻辑过长三种情况。

第一种情况通常在大规模的应用中都必然会出现，例如 Gmail 或者 QQ。当规模渐增，且我们必须“先完成登录验证再提供服务”时，登录逻辑的 CPU 消耗总量是不可能消减的。解决这一问题的基本方法是增加“登录服务”的物理服务器的数量，例如在 QQ 客户端上加入一个登录服务器列表，按照客户端的 QQ 号 Hash 到某个服务器去登录。又例如 Web 客户端（或其他没有处理登录位置能力的客户端），则可以考虑通过 DNS 轮循(16)来将请求投送到不同的服务器。

第二种情况常见于区域崩溃的恢复以及类似暴力攻击的时候。例如 2007 年台湾海峡的地震导致海底光缆断裂时，七条连接香港与外地的海缆中只剩下一条能维持有限度服务。在这种情况下，区域部署的系统（在软件与硬件上）都将受到访问峰值的冲击（访问浪涌）。即使不讨论这样极端的情况，在服务器维护进行重启后的短时间内，也通常会形成这样的峰值。而对于大多数在线系统（Online Service），首当其冲的就是登录服务。其原因便在于此前提到的：登录在逻辑概念上，就是数据单点的。应付浪涌的通常方法是“排队”，即在客户端或服务端添加有效的排队机制，将对登录服务的冲击控制在一定规模之下(17)。对于服务端来说，使用专用的队列服务，而不是依赖登录服务自身提供的任务池机制来处理也是相当重要的措施，专用队列可以负载更高，且对登录服务的处理能力不构成负面影响。对于客户端来说，提示用户“何时将会再次登录”是一个算法问题。事实上，我就遇到过同时打开某网站的十几个页面，而这些页面都卡在“短时间打开过多页面，请等待刷新”的提示页上。当使用浏览器的书签组功能时，这是常见的。这些页面采用的刷新值都是 5 秒钟，而十来个页面请求在 5 秒钟后“同时投送”到服务端的几率是相当高的。当你将这些页面想象成客户端连接，你就知道客户端延迟会导致请求最终像“共振”一样趋向同时抵达服务端。于是在上面的例子中，服务端与客户端都将同时处于 5 秒钟的饥饿等待与 5 秒钟后突如其来的请求爆发当中。

第三个问题比较特殊，通常我们不会主动在登录中加入过多的逻辑，但那些看似“无可拒绝”的产品需求则是例外。其中，“登录+权限”是一个最经常的假设，另一个则是“登录+验证码”。对于前者，事实上可以作为两个结点来进行设计，如果我们也将登录信息与权限信息设计为不同的数据的话(18)。对于后者，其计算的复杂性在于获取和核查验证码导致的外部调用（我们通常不会把验证码系统做在登录服务内），而这通常是通过验证码的预生成和本地缓存来解决的。然而还有一种特殊情况，即类似于在做第三方接入时，我们会异地验证用户的有效性。例如在系统中接入新浪微博账号，就需要远程调用其 Open API 并“等待”返回结果。第三方的计算复杂度或调用延迟将大大地拖慢登录服务的响应，进而导致 CPU 耗用很低而连接数很高（处于大量等待）的情况。从本质上来说，这也是因为在登录中加入了其他逻辑而导致的。

综合上述情况，如果系统中必然出现“（数据的或逻辑的）单点”，则应当一方面在单点服务内部入手，消减逻辑复杂度并剥离出更轻巧的服务，以减少单点逻辑的规模、提升服务性能；另一方面从客户端与服务端两面入手，尝试提供更合理的“请求－响应”逻辑，保障服务提供的体验；第三个方面则要努力尝试通过逻辑与数据的分布来消灭单点本身。

八

对于第三个方面，作为登录以及类似这样的服务，是否有可能设计为不是单点？数据镜像机制提供了一种可能，这也是我们要谈到的第三种以数据为中心的服务。其根本出发点在于通过后台的镜像与同步机制，将数据作为不变对象以保障逻辑对“数据的位置”的不知觉性。

图 3-22 比较了三种登录模型。

图 3-22　三种登录模型：单数据库方案（左）、增加应用服务器的方案（中）、增加数据镜

像的方案（右）

第一种是传统的单数据库方案。第二种是登录服务逻辑过大导致应用服务器资源紧张，同时数据库压力不大时，通过部署应用服务来提升系统整体处理能力的方案。第三种则是在第二种的基础上用以应付数据库压力的典型措施，即传统的数据库镜像方案。

几乎所有的大型数据库系统/产品，以及流行的开源数据库解决方案都提供了数据库后台镜像的能力，通常也提供增量镜像服务。这些系统以及这一典型解决方案事实上工作得很好，但是它也同时带来了大量的存储开销，并且不恰当地为“构建以单一数据库中心为基础”的系统提供了支持。

源于“单一数据库”的数据在总量上的变化，使得我们从传统的“集中式”向“复制式”发展变得越来越不可取。而另一种途径，即从“集中式”向“分割式”过渡的方案开始渐行渐显，也就是所谓的分表分库，如图 3-23 所示。

图 3-23　从“集中式”向“分割式”过渡：分表分库的方案

其中，最下层“（总）用户数据库”是一种出于安全性考虑的可选策略。分表分库决定了登录服务与各个子数据库之间存在某种强关系，例如：

□　采用同一的分布策略；

□　使用规则化的方法将分布策略写入到应用端（登录服务）的数据存储逻辑；

□　通过较大的存储过程来将应用端的请求转化为多个表之间的关联查询。

这种强关系带来了更大的系统架构负担，也就是说，我们必须在（包含上述三种方案在内的）多种方案中权衡。但无论如何权衡，都涉及在应用或数据库的逻辑层上的设计，以及由此带来的、不可控的开发规模。

反思这些变化背后的某些关联，“单一数据库”越来越明显地成为大系统灾难之源。而这是源于早期数据库应用（例如基于数据的三层解决方案）所带来的惯性思维。在我们面临的现实系统中，数据的性质在四个方面几乎在同时发生变化：其一，数据的碎片化和即时性；其二，数据向非结构化发展的趋势明显；其三，数据项次的量级，即数据笔数，随即时性以及系统处理能力的增强而快速增长；其四，数据与关联数据的不对等及其总量的规模化，例如一条微博可能附带一个视频文件。

这些数据性质的变化迫使我们在“数据库”之外寻求解决方案。其中，相对重要的原因是，数据库维护中对于“数据结构化”的要求以及实施基于数据库分布方案的代价，两者总的来说与重建一个数据存储系统基本相同。以上述的第四种情况为例，如果我们以前面提到的数据库方案来存储微博，那么附带视频文件的管理就要求对数据库存储与文件存储作统一设计；而当我们采用“镜像数据库”或“镜像文件存储”的方案来解决系统压力时，前端的内

容管理与存取接口，乃至于系统整体就几乎要重新设计一番了。

当我们的思考不再局限于“数据库”时，“将数据作为一个系统结点”的思维方式开始走向前台，如图 3-24 所示。

图 3-24　“数据结点”作为独立系统层次中的可部署对象

在这样的系统模型中，登录服务与其他种种服务并列在应用层，它们与“数据结点”一样作为可部署对象参与系统规划(19)。在这一模型中，系统的分布与并发策略建立在结点之间的 PD-IO 关系之上，而“备 1，…，备 n”这样的数据镜像，就只是出于对数据安全性的考量，而非是应付 IO 压力的临时策略了。

九

在这样的视角下，我们将越来越趋向于对系统的可组织性与组织方式的观察。以一个实际的、传统的系统架构为例：

对于整个系统（图 3-25），可以简单地描述为对“逻辑、数据及其交错关系”的求解（图 3-26）。例如我们将上述模型中的“P-D 关系”理解为某种数据层的提供方案，则可以得到如图 3-27 所示的模型：

图 3-25　传统的三层或多层系统架构

图 3-26　图 3-25 中架构的

图 3-27　分布式框架：对界面（对规则的封装）的思考

也就是说，我们将上述模型中的“P-D 关系”理解为某种数据层的提供方案，进而将服务对底层数据的存取变成一组分布逻辑，并将这些分布规则（rule）通过语言（例如 MapReduce 等）固化在对数据层的存储界面中——将界面理解为对规则的封装。如此一来，一种面向底层数据的分布式框架就形成了。类似地，我们也可以规划“P-P 关系”。例如我们可以讨论会话服务是否存有较多的业务逻辑、是否依赖实时响应等限制条件，并据此来选择合适的语言或执行环境。一种可能的情况如图 3-28 所示。

图 3-28　分布式框架：领域问题及其方案带来的思考

在这样的模型下，我们讨论的是在不同的层间所采用的领域方案，以及在这些领域交互界面上的可行性、安全性与系统代价。在不同的层间，由于所关注的系统性质不同，因而候选的标准与工具也不同，其数据的格式与存储要求也存有差异，但总体来说，是对 PDIO 四个方面的综合考虑，例如逻辑（P）的复杂度、数据（D）的量级、IO 的频度等。

除开这些在分布、部署、调度等技术细节上的分析与选择，我们要讨论的将是这些层间的规划与层间关系的模型，以及如何通过系统化方法来实现这些层间（也就是领域间）的协作开发。而这些将部分会是本书下一篇所涉及的内容。

————————————————————————

(1) 事实上，业界现在又把这两个学术化的领域统一称为“云”，包括云计算、云存储等具体的解决方案。关于数据或运算的规模也存在许多其他细分的领域和混杂的概念，例如早期的网格（Grid）与现在的大数据（Big Data）。无论如何，这些都只是分布问题在业界的具体表现而已。

(2) 就目前来说，通用应用开发语言主要是指面向对象语言或结构化程序设计语言，例如 Java、C#，以及 C 语言等。

(3) 因此，在缺乏这样的需求——例如在一个小一些的、跨领域特性并不突出——的系统或应用中，要求实施“面向接口设计”是一件相当多余而又学术化的事情。

(4) 另外，我们也可能通过静态数据与增量数据、本地数据与远程数据等分类法来区别不同的数据。我在这里采用“结构化”这一分类法只是一种选择，仅为了说明常见的问题，而非一个强制设定。

(5) 这涉及数据的全部三种性质：标识、值与确定性，以及由确定性带来的分布、状态等性质。

(6) 两者除了抽象视角的不同，在实现上通常也有差异（这里的服务与结点主要分别基于 Java 和 Erlang 中的概念）。

(7) 在定制的 HTTP 客户端中，也可能将验证信息直接放在 HTTP Head 中，例如超星浏览器等使用 HTTP 协议的软件。但是这在通常的、基于浏览器的 B/S 应用中很难有通用性（如果非要这样做，可通过浏览器插件来实现）。

(8) 在这里提及这种可能性，是因为某些情况下，在这个位置上会验证客户端的来源（例如 IP），或强制使用 Get/Post 请求，或判断是否有文件上传等。这些措施通常通过添加服务器模块（Module/Filter）并通过配置文件来实现。

(9) 也可能存在其他的约定，事实上有些人认为将 SessionId 放在 Cookies 中更加有效。但在这里，我们是假设要同时验证“HTTP 请求的行为（Action）+SessionId”两项，因而可以放在 URL 中并通过一次解析来得到完整信息。

(10) 基于 Linux 系统在文件存储上的优势，“Path/File”也可以被视为“Name/Value”或“Key/Value”数据存储的一种，并且通过文件路径以及文件名（filePath+fileName）也可以实现简单 Hash。事实上，这些面向存储的数据处理，比我们通常在应用程序中、发生在函数调用界面上的数据更为常见。在系统组织中，不要仅仅将应用程序（及其逻辑）视为系统的组成构件。例如“文件服务器”就可以通过我们在上一小节中提到过的基于部署视角被理解为数据“结点”。这一结点的界面是“操作系统的文件存取接口”，包括 System API 或 SMB（Server Message Block）协议下的共享访问接口等。

(11) PD 模型描述处理（Process）与数据（Data）之间的关系，通常用于对计算范式的描述。

(12) 我们讨论过，这正是函数式语言的优势——将逻辑作为可迁移对象通常是基于运行机制来保障的、具有逻辑自身的不知觉性，而命令式语言则难于实现这一点。但看起来，我们在这里所面临的又似乎是典型的“数据不变”的系统环境。所以语言范式之于系统模型，是两种语境。

(13) 该图是对 Peter Van Roy 的“主要编程范式”一图中部分内容的重新表达。参见：www.info.ucl.ac.be/～pvr/paradigms.html

(14) 从概念上来说，数据全集 x 包括指示它自身的状态 Sx；也就是说，Sx 是 x 之于时间的信息。

(15) 这里的“同一个会话数据”即是指数据确定，并强制要求使用该会话数据的逻辑不得有 update 操作。

(16) 通过配置 DNS 服务，使相同的域名解析为不同的 IP，对应于不同的物理服务器。

(17) 所谓“一定规模”，即是在服务上线前对“响应数/秒”的理论计算，也需在服务期间进行长期跟踪调适。运维与监控是大型系统中相当重要的组成部分。

(18) 只不过它必然带来二次交互，或在登录中发生向权限服务的远程调用。但一旦将验证与权限设计为两种数据，则这两种方案替代的成本极小。我在实践中，就曾经要求在开发中使用一次交互（在登录服务中向权限服务远程调用），而在上线时使用二次交互并分别部署登录与权限服务。

(19) 这样的思维框架是将所有的可部署对象作为结点，而并非唯只是数据结点。

附录一

“主要编程范式”及其语言特性关系(1)

一

Peter 将一个最重要的概念“state”引入进来。而这个 state 也是 Peter 对语言进行分类并考察其变化的主要依据。但 Peter 所使用的 state 概念以及专用名词“cells”都相当地令人困惑。所以在这个图的补充说明中，Peter 对此专门做了解释：“状态是记忆信息的一种能力，更精确地说，是及时存贮值序列的能力。”

你觉得这个概念像什么？对了，的确，非常像是“变量”。事实上，状态在编程核心概念的“数据-逻辑”抽象中，表明的正是“数据的可变性”。也就是说，数据的可变性表现为状态。更进一步地，当数据被命名时，它称为变量或常量；当数据未被命名时，它成为游离的、无名称的、时序含义的存储单元，即“cells”。这也是 Peter 使用“cells”这个专用名词的原因：在本图的讨论中，需要从“变量/常量”这样的概念中，剥离掉“未命名或命名的、确定的或非确定的，以及串行的或并发的”这三个方面的性质。

当不考虑一个存储位置上的命名特性时，它就既非变量或常量，也非某个确定的运算对象（例如“对象”等高级的抽象概念），而只是一种更加泛义的“状态”；同时存储这个状态本身的事物，由于没有位置、时序等概念，所以被称为“cells”。

二

《程序设计语言：实践之路》这本书解释过命令式语言的本质特性，即用算法改变数据。如果用两个以上的逻辑（例如两行代码）去影响同一个存储位置（cells），使它的状态改变，并最终在该 cell 产生运算结果，那么它就是一种命令式语言。

再简单一点（但没有上面这样严谨）地说：在程序中不断地重写变量，变量值即是程序的最终结果。所以在本图中，Peter 把这个衍生关系表达为：命令式=纯数据+算法+状态维护。

无论是在串行还是在并发的编程中，命令式编程范式对“状态”的理解都是：共享状态。在串行（例如单线程）的编程中，状态是时序相关的。因为不断地重写“状态（数据/cells/变量）”，所以前一行和后一行所面对都是同一个共享状态的不同的值/副本。在这个过程中，正因为状态与时序相关，所以前一分钟与后一分钟的状态是不确定的。但是在同一时刻，这个状态是确定的。

与上面相类似地，在多线程中，同一时刻，不同线程也将面临这个值/副本。但正是因为多线程（并发）中，线程 A 与线程 B 对于同一个 cell，在同一时刻所得到的状态也是不确定的——我们可以假想为多核 CPU 在对同一个内存地址读写（于是就出现了我们所谓的“同步”问题，进而也就出现了“锁”的问题）。所以在这个分支中，当加入“线程”概念之后，新的编程范式全都变成了“可观测的非确定性”为“yes”的情况。

三

我们显然可以发现，问题出在由于多个线程都在“写 cell”。在命令式的解决方案中，采用的方法是“加锁”；持锁存取的最经济的方法之一是“多读单写”，即保证同时只有一个线程能“写 cell”。

但是这给应用带来了负担。如果一个应用程序有多个线程（分布或不分布在多个 CPU 核上），在它们都要读取同一个 cell 而又有某个线程要写该 cell 时，那么大家就都要被挂起来，直到这个写操作完成。整个应用程序在 CPU 使用（或者说效率）上就大大打了折扣。

如果这只是一个桌面程序（例如记事本），大概没人会说什么。但如果这是个服务器程序（例如 WWW Service），那么整个网络、所有的会话就都处于等待状态了，但同时，服务器的 CPU 占用可能会远远小于 1%！

解决问题的终极方法，就是不解决这个问题。既然写 cell 带来了问题，那么我们就“不写 cell”。我们由前面所有讲述的内容开始倒推，问题根本是由“命令式语言”这个编程范式本身决定的：用算法改变数据。

所以我们回到了原始的问题：如果算法不改变 cells（数据/状态/变量）呢？

————————————————————————

(1) 节选自博客文章“‘主要的编程范式’及其语言特性关系”（2009 年 10 月）。文章对 Peter Van Roy 的“主要编程范式”一图进行了解读，Peter Van Roy 的原文参见：www.info.ucl.ac.be/～pvr/paradigms.html

附录二

继承与混合，略谈系统的构建方式(1)

面向对象系统有三种对象的继承方式，即原型、类和元类。这三种方式都可以构建大型对象系统。在后续讨论之前，我们先在名词概念上做一些强调。所谓“对象系统”，是指由“一组对象构成的系统”，这些对象之间存在或不存在某种联系，但通过一些规则组织起来。所谓“面向对象系统”，是指以上述“对象系统”为基础延伸演化的系统，新系统满足前对象系统的组织规则。

所谓“对象系统的三个要素”——继承、封装与多态，即是上述组织规则的要件。孟岩同学从 C/C++出发(2)，从另一个侧面谈论对象系统，所持的观点我相当认可。他指出，“对象范式的基本观念中不包括继承、封装与多态”，这一观点有其确切的背景与思考方法，值得一谈。

我们在这里要讨论的是“对象系统”，即对象是如何组织起来的问题。在这个问题上，组织规则之一就是“继承”。JavaScript 中基本的继承模型是原型继承，其特点是“新对象实例的特性，复制自一个原型对象实例”。Qomo 以及其他一些项目，通过语言扩展的方式，在 JavaScript 上添加了类继承的模型，其特点是“对象构建自类，类是其父类的一个派生”，这里的“派生”与“特性复制”有潜在的关系，即子类的特性也复制自父类。正是由于“派生”其实是“特性复制”的一种形式，所以事实上 Qomo 中的类继承是通过原型继承来实现的，因为原型继承本质上也就是“特性复制”。

无论是原型继承、类继承还是这里没有进一步讨论的元类继承，继承的最终目的都是构建一个“对象系统”，而不是“系统”。这一个措辞上小小的区别，有着本质上的深刻意义，这也是我提及孟岩的那一篇文章的原因。通常由“继承”入手理解的“对象系统”其实是静态的，以至于我们在面向对象系统开发的最后一步，仍然需要框架来驱动它。例如 TApplication.Run()，或者类似的 new Application()等。继承所带来的，主要仍然是指对象系统的组织性，而非其运行过程中的动态特性。

于是我们通过更多类或其他对象系统，来将一个系统的动态特性静态化。例如将对象之间的交互关系抽取出来，变成控制类。我们做这些事情的目的，仅仅是因为我们约定了对象系统的组织规则，要面向这个对象系统开发，也必然满足（或契合）这一组织规则。组织规则限定了我们构建系统的方式——继承、封装与多态，这在一定程度上说是“对象系统构建”的一个方案，并非“系统构建”的方案。而孟岩在文章中所讨论的，正是“系统构建”的问题。所以孟岩提出两点：

□ 程序是由对象组成的；

□ 对象之间互相发送消息，协作完成任务。

其中第一条，是对象系统的基本特性，是谓系统成员；第二条，是对象系统如何演进为系统的特性，是谓系统通信。一个系统的约束，既包括其成员（以及成员的组织规则），也包括成员间的通信。

————————————————————————

(1) 节选自博客文章“继承与混合，略谈系统的构建方式”（2010 年 12 月），文章讨论了对“基于对象系统进行系统构建”的认识与实现。

(2) 参见孟岩的博客文章“function/bind 的救赎”：

http://blog.csdn.net/myan/article/details/5928531。

附录三

像大师们一样思考——从“UML 何时死掉”谈起(1)

算盘用了几千年，谁问过“算盘为什么能算东西”？算珠、进位、栏，这些东西是不是基本的存储结构？用算盘的“我们”，是不是计算单元？珠算表是不是运算规则？那些珠子表达出来的“0～9”的排列，是不是输入输出的界面？

“我们+算盘”就是一个完整的计算系统。这样的计算系统的完整性，图灵用了一个假想加以说明。图灵不过是用一个假想描述了一个事实，而这个事实“看起来”能被机器实现。于是，我们的计算机时代就开始了。

图灵是否证明过“大笨象吃意大利面条为什么是一个完备的计算机系统”呢？不，最初等的问题，往往最难于证明。往往，他的证明过程，或应用过程，只是触发了一个想象。

对计算机根本问题的思考，许多会追溯到哲学思想层面。IOPD 和 PDIO 的问题、“算法+数据结构=程序”的问题等，就属于这一类。还有一些会追溯到人类行为学、语言学等层面，例如语言、语法、语义，以及像有没有语用这样的问题。大多数时候，真正推动计算机发展的，不是对具体问题的推理求解，而是对问题本身的抽象。在 Dijkstra 的叙述中，抽象更像是终极武器。按照 Brooks 的观点：

数据的表现形式（数据结构，抽象的结果之一）是编程的根本。

而按照 Dijkstra 的引述：

引用未解释过的名词阐述公理或定理和作用于未解析过的操作数的（命了名的）运算两者之间有着某种平行的相似性。

无论如何，我们“做一个计算机”，原始的目的不外两个：其一是“让它计算数学”，其二是“让它像人一样思考”。请注意，我的确是说“计算数学（即算数）”。数学是人类的理论学科，“怎么算”以及算的内容等，都是由我们自己设定的。而计算机的能力，只是计算“数学”这个它未知的对象而已。事实上，我们现在讲的“命令式”、“函数式”或“说明式”，只是我们为计算机设定的“最基础的运算方式”。在这个“运算系统”中，“数学”并不是最初设定的。

命令式如何计算，函数式如何计算……诸如此类的问题了解清楚了，我们对这类语言也就了

解了。至于什么高阶函数（higher-order function）、克里化（Currying）、延续（Continuation）或发生-迭代器（Generator-Iterator）之类，那已经是具体语言的表象，而非“这一类语言”的本质。举例来说，JavaScript 1.5 还没有实现过“生成器对象”（Generator Object），但并没有人否认它是函数式语言。反过来说，“Generator Object”原本就不是函数式语言的必备要素。

LISP 表达了函数式语言的全部“必备要素”，然而 LISP 七个原子运算也只是针对 LIST 这个结构抽象来说的。对于一个“（顺序的）表”，这七个原子运算是必需的，而对于另一个“（关系的）表”就未必如此了。所以某些原子运算，也不必放在函数式的必备要素中。像 LUA 这样的函数式语言实现方法的出现，也证明了这一点。(2)

那么函数式还剩什么？

要真正理解函数式的秘密，是要一个语言一个语言地学习下去吗？是要一种运算法一种运算法地学习下去吗？我们听完人家说“持续”，于是就开始了解持续，而没有去追问持续为什么出现在函数式里面，或它是不是函数式的必备要素，又或是函数式运算系统的自身的“问题”。我们正是迷失于种种语言和概念的表象，而最终没能像大师一样去思考“计算机不过是大笨象吃意大利面条”这样的抽象层面的问题。

我们要改变的是思想，我们要增强的是能力。大多数人只是增强能力，而不改变思想。这就是我们大多数人不是大师的原因。

————————————————————————

(1) 节选自博客文章“像大师们一样思考——从‘UML 何时死掉’谈起”（2008 年 10 月），文章是在与“UML 之父”Ivar Jacobson 先生座谈后的反思。

(2) LISP 的基础数据结构是索引数组（表，LIST），而 LUA 的基础数据结构是关联数组（表，MAP）。

附录四

VCL 已死，RAD 已死(1)

当浏览器成为普通用户使用计算设备（包括移动的、桌面的、嵌入的等）的首选时，它便隔离了操作系统与 Web 环境下的 UI。我们没有在任何地方看到一项要求说：一个 Page 必须要

有一个跟浏览器 Toolbar 风格相同的工具条，或跟窗体风格相同的菜单。从本质上来说，是浏览器的便捷与普众，催生了 B/S 结构下的应用和服务开发。而这样一来，桌面原生的客户端就不复存在了，C/S 结构的应用渐渐地开始消失，除非在客户端存在较大的运算、逻辑或对计算环境的控制。

重量级的客户端软件越来越少，因为从根底上说，人们不喜欢用复杂的软件。领域的边界，从浏览器编程界面退缩到网络界面。也就是说，浏览器端（Web 客户端、B 端）的开发人员不再要求“能够调用 Win32 API”，而是要求“能够进行网络交互”。而当这一阵线真正推进到面向 socket 的二进制编程时，操作系统就被从这个体系中切割了出去。

Flash Socket 以及 HTML5 中的 HTML Socket 带来了这种趋势，这种趋势让微软措不及防。一方面 Sliverlight 还在为 Flash 仓促应战，另一方面 IE+JScript 的结构尚未完成六年来最大、最根本的变革（IE8、IE9 或 IE10）。然而即使如此，一个如同操作系统一般庞大的 Web 领域，已然形成。在这个领域中，微软仍在第一战线，且树敌良多。

当我们把 Web 看成一个像操作系统一样的产品平台时，“程序员”便成为产品生成链条中的一环，程序员文化是被重点考虑的对象，但不是全部。包括平面开发人员、设计师、架构师、部署专家、行业分析人员等在内的团队模型必然会建立。小型的 XP 团队仍将存在，但这取决于应付的系统规模，以及在纵向切分上同质性的多少。

横向切分将出现在浏览器端开发的整个过程中，这不但是指整个 UI，还会有 UI 过程中的各个细节，例如框架、数据交互、网络界面等。在这一过程中，纵向切分依然会成为补充。例如将网络界面与数据交互并成一个独立的部分，交由 Flash Socket 来实现，或交由独立的 comet 兼容层来实现。但更确切地说，横向分层仍会带来更细分领域的繁荣，例如 JSON 或其他微数据格式，以及其他基于 Socket 或 http/https 进行交互的二进制数据格式将成为专门的研究领域。

这其中的原因是，在 B 端带来的领域必然扩大到一个无法通过纵向切分来一次性交付的地步，因而必然在这一端出现更细化的横向分层。从经验来看，当一个领域足够成熟时，就意味着它可以接受横向分层了，正如现在的桌面作为一个领域，可以接受 UC、UCC 以及 NDA 等(2)更为细化的分层一样。

横向切分是领域合作的模式，这也导致横向切分与金字塔式的管理模型结合时，会存在多领域专家汇聚在金字塔顶端的情况。当这种情况出现时，就需要更高的决策层来应对，这也意味着决策层需要更多的经验和能力。当然，我们仍然会失败，因为即使我们把系统先纵后横地切成网状，我们仍然面临总体规模上的复杂性。同时，管理规模的扩张，也导致我们的成本增加，周期拉长。

所以如果你不是做 3～5 年的规划或者常常被人垢病的“太空项目”，那么你不需要考虑一个问题的全集。你需要关注的是，在某个具体项目中，是否更合适于某些层面的横向分层，并且有意识地培养该层上的开发人员与相关角色。

我认为可以有颠覆性的思想，但从来不指望颠覆性的变革。所以能同时兼容横向分层的后 RAD 时代是漫长的，不过即使是三两年，我想，在 IT 业来说，也算得上是漫长的了。

再接下来，更为迎合这种面向领域组织团队并开发的工具便会出现。但这种工具不再期望整合各个领域的实现技术（注意我不是说“开发技术”），而是提供领域间的交付标准，或者更为直接地提供交付物。更多领域专精的公司受到关注（例如现在的 Macromedia），大厂商开始并购更多的专属领域的公司，以整合他们的业务。更大的平台化产品会出现，远程的、分布的、可迁移的运算理论和解决方案被普及；而与此同时，更细分的领域带来了更多的专属工具和专精人才；项目的整体规模扩张，并由多个团队来实现（像工程的包工队一样）；而单一团队中人员结构更为复杂，但团队规模仍然被保持在 10 人以内，以 15～20 人为上限。类似于测试等技术，将会作为领域而出现，类似于现在建筑业的工程验收与监理也会出现。这些专属领域仍然有它独特的标准、技术与工具，并提供独立的交付物。

对于整个工程来说，RAD 彻底死掉了。也许类似于建筑行业中的沙盘/模型制作公司会出现，并成为产品过程中的一环，但再也没有了在多个横向切分层面上贯通的 RAD 工具。基于原型理论的 RAD 过程，以及迭代的 RUP 会慢慢地退出去。因为如同没有人能够提供一个楼房建筑的迭代过程一样，工程的复杂性已经让人远远不能控制单次迭代的成本。在这种规模级别之下，对层间界面的控制决定了系统的稳定性，以及能否持续开发。因此架构师在全程成为必须，而且架构的职责将更为细分，例如精密到一个数据 IO 界面，可能都需要特定的架构师来确认和论证。同样地，局部的失败或失效将在系统的更早时间内被发现和验证，返工在局部成为常态，但在全程却不复存在——或直接地让整个工程失败掉。

工程越来越依赖于经验，以及由经验带来的技术模型。例如我常常提及的塔楼式建筑，并不是某个工程学家或领域专家头脑风暴的产物，而是在实践中总结的经验。工程的可复制性增强，而复制规模和环境则更趋简单（过于复杂的规则与界面是难于复制的），这一切依赖于横向分层的界面上的简洁性。

模块的复用与重构依然会长期存在，绝不会消亡。更细的复用粒度必然从现在的对象扩展到业务组件复用，但本质上仍然是以无业务逻辑存在的基础对象复用为前提。用户界面（UI）作为组件将会在很长的历史上出现、消亡与重现；更多的层间界面（interface）将以协议标准的形式出现。而在各自独立的层上，基于层间界面的逻辑组件将成批地涌现，“数据”仍然是它们的唯一约束与编程本质。

我们会有对计算模型的新的尝试，但本质仍与如今一致。函数式语言将会走向前台，脚本语言成为领域间的粘合剂（尽管现在在某些领域间已是如此），确定的语言将在确定的层次上大放异彩（或这也是领域语言的一个代名词）；同时掌握关联的多个层次上的开发工具的专家，以及领域专长的、与程序无关的专家将成为热门。对于整个体系来说，前者主要是实现具体的业务逻辑，而后者则专注于数据定义。不过，可以想见的是，没有多少人会特定用 XML 来作为这些数据定义的载体，专属的数据格式将是层间交互的首选。

最后，从工程实践的角度上来讲，二十年之内，我想不会出现一本名为《软件工程之营造法式》的书。当然，某些噱头制造者或纸张贩售者的吹嘘除外。

———————————————————————

(1) 节选自博客文章“VCL 已死，RAD 已死”（2008 年 12 月），文章从用户界面开始，最终推进到对象系统界面与组织方式的思考。

(2) UC：UI/Client；UCC：UI/Control/Client；NDA：Network/Data/Application。

引言

架构师的思维

一个人太不切实际，我们称之好高骛远；一个人有眼界视野，我们称之高瞻远瞩。同样是高、远，为何描述着两种完全不同的人？因为前者的好、骛讲的都是追求，而后者的瞻、瞩，指的却是具体的行动。当我们把高远的目标只作为一种追求，而不付诸于实践的时候，就是不切实际；当我们把它变成“时时顾看”这样的行动时，我们就渐渐地变得有眼界视野了。

所以志存高远并没有错，只是要切忌不务实。这里的“着眼于高远”，便是架构师的基本修养，而几乎所有的架构思维，都从这修养中来。

就架构来说，“高”就是指空间上的可拓展性，即系统的复杂性是否可以通过组成部件的增减来解决；“远”就是指时间上的可持续性，即系统的规模是否可以划分为多个时间阶段来实施。以软件架构为例，在讨论系统——这一架构目标的属性时，架构师可能关注的话题包括性能、可用性、可靠性等十余种，我们可以通过高、远两个维度的思考将它们大致地分类，如图 4-1 所示(1)(2)。

图 4-1　对架构师可能关注的话题的分类

如果我们说，这样的图是没有意义的，因为它对一个工程的具体实施毫无意义，那么这是项目经理的思维；如果问这样的图是如何以及用什么样的工具做出来的，又或者讨论填以什么样的颜色更为漂亮，那么这是程序员的思维。但是，如果我们问：在这个图的形成中，我们做了什么？那么，这就是架构师的思维了。

如上这些谈论，总的来说包括了修养与思维这两个方面。其论述为：

（1）我们做了一个语言文字中的高、远与架构思维中的高、远的比拟；

（2）我们对架构思维中的高、远进行了明确的定义；

（3）我们将既有的架构方法置于上述定义，并尝试消化其中的冲突；

（4）我们确定上述思维的结果，是一种架构产出；

（5）我们通过对比找出几种思维模式的差异，并确定上述过程，即是架构师的思维；

（6）我们通过回顾这一过程，证明架构思维过程的有效性：产出上述的架构。

在上面，我们反复地运用架构思维，得到了两个主要的架构产出，其一是“架构师的基本修养”，其二是“架构师的思维过程”。在这个过程中，不同阶段我们使用了不同的思维工具，如表 4-1 所示。

表 4-1　架构师在思维过程中使用的工具

任何一个优秀的架构师都有自己独特的思考方式，这决定了他如何抽象系统，以及如何“创造性地”设计与构画这个系统。例如，我们一直在讨论的“架构思维”——这样一个内在的系统与规则都是未可知的新东西。对此我们没有现成的、成熟的词汇去描述它，因而必须构建一个抽象系统，或映射或重现这个“架构思维”，进而阐述清楚它的架构与逻辑。

在这个过程中，我们需要三种能力：概念抽象能力、概念表达能力和基于概念的逻辑表达能力。我们已经展示了概念抽象能力，即上述步骤中的第 1～3 步；概念表达能力，即图 4-1 与表 4-1；概念的逻辑表达能力，即上述步骤中的第 4～6 步，以及至此你所看到的全部过程。

————————————————————————

(1) 这些话题引自 Software Architecture in Practice、Evaluating Software Architectures: Methods and Case Studies 与 Java Web Services Architecture，但这里不讨论这些话题是否完整或者必须——这与“如何思维”没多少关系。同样的原因，不必讨论该图是否划分得准确，或是否正确地将某些特性置于交集中。

(2) 制作类似图例以表达思维结果，也是架构师的基本能力之一。

第十章

你所关注的系统

系统，是对架构师所面对对象的基本抽象。架构师对系统的认识过程、方法与结果，决定了他如何理性地架构之。本章将讨论从现象到本质地认识一个系统的过程。

认识系统不是架构系统。认识系统将致力于将系统中的核心概念抽象出来，将核心逻辑梳理出来，将核心问题（关系、依赖与冲突）揭示出来。但是架构系统的目的正好在于通过对概念与逻辑的映射来消弥这些核心问题，使核心问题对其外在（例如用户可见的产品）不构成明显的影响。

架构是一个过程。既然是过程，必然有起始与终的。本章将架构过程设定为一个以架构意图驱动的模型，讨论其起始问题中的架构意图的产生与确定，但不讨论“架构的终的”这一问题。

本章所讨论的系统是一个泛义的概念，并不具备规模性的含义，因此它与第二篇和第三篇中的指称规模的“系统”并不是同一概念。同样的原因，本章所讨论的架构也是泛义的。

第一节　了解系统的过程

一

我常常设想一个场景，这个场景是如此的简单，以至于只能用这样平白的文字来叙述：

在一间黑暗的屋子里，突然有光线照进来，你发现：什么也没有。

我迷恋于这一场景的原因在于：它表现了我们认识一个系统的、最初的、一刹那间的感受。

是的，我用到了“感受”这个词，因而我必须先讨论什么才是你的感受。

在我写这段文字的时候，新闻中正在播报土耳其发生了里氏 7.2 级地震，提及到死亡人数近 300 人，在背景画面中闪过了一位中年男子抱着一个受伤的小女孩的映像，男子显得有些紧张，嘴里在说着什么，而小女孩则一脸惊恐，无助的眼神投向摄像机镜头。我从这一画面及其背景映像中获得了非常多的信息：从地震等级到死亡数字，从环境的混乱到小女孩的伤痛……这一切建立了我对这一场景完整的、刻板的信息。我如同一页纸，被书写了一个个的概念名词，或者绘制了一帧帧图像。

如果我真如一页纸，或者如同计算机一般，是一个信息的载体或渠道，那么上述的一切将是

我对这一事件的全部了解——或许会细致更多，但本质上是没有区别的。

但我悲悯，我想哭泣；我有一种冲动想去帮助他们，帮助这些正在苦难中的人们。无论如何，那一时刻，我总是想为我所接受的信息做出一些反馈的。但我反思这一“反馈”冲动的原始愿望，发现它们都无一例外地将出处指向一个词汇：感动。是的，我们的新闻媒体，也包括那些广告宣传所做的，都是试图去“感动你”或“感染你”。当它们达到这一目的之后，我们——作为这些信息的受体，就会不由自主地有反馈的冲动。最终，如何把握受众的情绪，进而把握这种反馈冲动的时间、方式等，就是新闻与媒体的奥义了。

如果我只是一个有反馈机能的受体，就像我们制造的某些力反馈的、视觉反馈的机械装置，那么上述这一过程将是我所有行为的极致——或许在人工智能方面会更复杂一些，但本质上也是没有区别的。

但是我还觉察到一种疼痛，如同小女孩一般，我的手臂感觉到她伤处的痛楚；又如同中年男子一般，我内心充满了不安与焦虑，我急切地期望为小女孩找到医生；又如同整个场景，我很快地陷入了巨大的悲伤与恐慌，我为每一个人的安危担忧起来……

这才是感受：以体察之，感同身受。

二

让我们回到那个黑暗的屋子，设身处地去感受一下这个屋子的存在？

有光线、明暗的边界、屋顶、四壁、砖石、门；有窗、窗格的木条、木条上的纹理、纹理中扭曲的形像；有灰尘，地上的灰尘、空气中的灰尘、窗台上的灰尘、窗格上的灰尘，无处不在的但并不厚密的灰尘；有空气，光线中的空气似乎要清新一些，而墙角的空气则显得阴暗潮湿，（我俯下身）地面的空气好像透着丝丝凉意；我在屋中跑跳了几步，嗯，不错，看起来今天会是不错的一天——尽管我还没有推开门，或者我也并不知道这是不是一个锁死的牢房……

是的，这有点散文或小说的笔法，总之看起来像是文学作品中的桥段。正是如此，我们作为“计算机专业人士”的日子太久了，我们对太多的事物有了理性的认识，而缺乏感性的认识。正因为我们忘却了这种“感同身受”地了解事物的方式，所以我们对这些事物的认识流于浅表，流于那些有数字个数、形体大小、边界棱角或者演进逻辑的判断推理当中。我们忘了一个“系统”是可以去知道、了解、感知，进而感受的。

我们把对系统的观见与解说当成一种理论，这种理论称为“需求分析”。而我们在一定程度上忘记了，我们所谓之“系统”，并不仅仅是模块的组成，也是一种外界——之于这个系统——的认知(1)。

三

认识与感受是不同的，我们接下来讨论“认识”。

一个人认识到另一事物，包括对这一事物的两种了解，其一是它的外在，其二是它的内在。例如说认识一个人，男女老少、发肤形貌等这些是外在特性，品性德行、气质教养等这些则是内在特性。“认识”的这两点要求，无论要了解的对象是复杂的人还是简单的石头，都是必要的。

我们获取知识有两种手段，是所谓“知得”与“识得”。在本书的第二篇中，在讨论抽象这一主题时，我们提出过这两种手段(2)：当我们知道张三，却不知道它的形貌时，是知得，大多数图书馆知识是知得的；当我们亲触这个事物(3)，却不知道它是什么时，是识得，大多数野外考察知识是识得的。

从动词角度上来说，认识是识得的具体方法之一。认，是指记认；识，是指辨识。

记认作为一种方法，可以与我们讨论过的曹冲称象与刻舟求剑这两种实践联系起来。它们在船体上刻的标记，都是记认的一种形式；同理，我们后来讨论到的 HASH，以及全文或数据库检索中用到的关键字，也都是记认的形式。作为实践者，我们大多数时候是在讨论“某种记认的方法”，而未能追究：在认知理论上，这种记认的可靠性及其依赖的条件。而忽略这一点，就会产生一些似是而非的方法，例如失效的刻舟求剑。但是，失效并不是无法容忍的，例如 HASH 应用中存在的命中率问题。所以记认并不是准确无误的方法，实践中只是在寻求这种方法的背景限制并进一步控制误差而已。

辨识的一个基本含义在于分辨出差异。如果找不到差异，那么所有的事物也就混沌一物，无从辨识，也无从获得它的知识了。具体来说，辨识也可以分成两种方法：其一是识别，其二是分别。

识别依赖于我们对事实的直观了解，在一定程度上，这是与我们的感觉器官相关的，例如听见的、看见的或者闻见的等。识别是我们人类建立对自然界的知识的最基本而又最丰富的方式。大多数情况下，我们不会去考虑我们如何从树林中识别出一个新的树种，或者如何从风声中听到猿啼，这基本上被我们视为本能。我们的这一类知识构建行为，大抵在于为这个新树种命个名称，或者此前便已了解怎样的声音才是猿啼。

然而识别是不可靠的。它首先取决于生理机能本身的可靠性，例如一个红苹果，在正常人与色盲症患者的认识中，就并不相同。其次它还取决于既识的持续可靠性。我称“基于识别所构建的既有知识”为既识，称基于既识而识别为持续性。例如，某人此前听过猿啼（并确认正确），当他再听到某种啼声时，识别为“这是猿啼”。但后者并不一定是持续正确的。后者的正确性涉及三个具有递进关系的问题，其一，猿啼是否必须是一只真实的、自然界的、实体的猿的啼叫；其二，若否定其一，则需讨论非真实的、模拟的猿啼在多大程度上能称为猿啼；其三，若肯定其一，则需讨论如何同化个体猿的声音差异，以使得“（任意）猿的啼声”总能被识别。这三个问题的提出，事实上说明我们“基于既识的识别”是不可靠的。

分别则相对复杂一些，它建立于一个观察的角度、切面，或者依赖于某种参照。以（概念性的）观察角度为例，地上散落的核桃，A 可能将它视为“一些核桃”，而 B 则认为是“三堆核桃”，这是整体视角与局部视角的差异；又例如，同向同速的两辆火车之间的观察是相对静止的，这就是参照选择带来的一个结果。通常，“数”这一抽象，是我们能加以分别所依赖的核心概念。例如，核桃的个体与群体，以及火车的速度，都是我们对观察对象先进行数

值化，再加以比较，最后得到的知识。

分别是可靠的吗？答案仍然是否定的，千人千面是一种理想状态，现实往往是一人千面。例如一个人早晨显得慵懒一点而中午就亢奋些，又例如对于同一个人，A 认为他和善，B 则认为他隐忍。分别的问题在于比较所需的角度与背景不同，以及不同人对于抽象概念的理解有异——如你所见的，基于“数的值”的分别往往准确一些，是因为人们对于“数”这一抽象有着大抵相同的理解。

四

我们讨论上面的认知理论，其实是在讨论我们建立“知识”的具体方法。然而如上面讨论的，我们从一个系统中获得的知识因人、因方法而不同；既便是相同的方法，由于其实施者的不同以及方法（本质中存在的）误差，也会不同。这就是作为架构师，任何两个人都不可能得到相同的架构结果的根本原因。所有的最终架构都是在实施过程中的调和，以及某些决策者、决策机构的“决定”。

大多数的外在特性是容易从系统中辨识出来的。例如，我们要做一个办公系统（OA，Office Anywhere），那么我们可以肯定几点事实：

（1）这一系统总是某些办公室成员使用的；

（2）这一系统总是提供上述人员的日常工作所需的功能；

（3）这一系统既包括对现实工作的映射，也包括一些试图改变现行工作的电子化需求。

这几点事实显而易见，是由系统本身决定的(4)。我们可以因此找到一些系统的组成部分：

（1）观察办公室成员的工作，所以需要邮件、日程、考勤、审批等功能；

（2）考虑到电子化管理，所以需要新闻发布、消息通信、文件管理、讨论区等功能。

据此很快我们就可以描述出这一系统的架构，如图 4-2 所示。

图 4-2　通用办公系统架构 v0.0…1

但这些只是一些共性的功能，也就是大家都需要的。随着你对办公室成员的调查进一步地展开，你必然面临一些特定的需要，例如：

（1）人力，即档案、招聘、培训；

（2）营销，即客户、活动；

（3）经营，即资产。

据此我们进一步补充这一系统的架构，如图 4-3 所示。

图 4-3　通用办公系统架构 v0.0…x

回溯我们对这些“功能性模块”进行分类的依据，我们可以为这个系统的三个主要部分命个名，分别为“日常办公”、“电子化管理”与“特定业务”：

几幅架构图的演进关系并不难理解，但有一点点差异：图 4-2 至图 4-3 的标题中的版本号是“v0.0…x”，而在图 4-4 中却是“v0.0.0.1”。更深层次的问题是：何以认为前者连“一个架构的阶段性版本都算不上”，而图 4-4 却可以称为“一个最最最初级的架构版本”？

图 4-4　通用办公系统架构 v0.0.0.1

我们可以依赖种种视角对系统加以观察，并添加种种分类依据来得到前两幅图所示的“v0.0…x”版本的架构。但是，这些“识别”与“分别”的方法，无助于你得到“v0.0.0.1”中的几个关键概念：日常办公、电子化管理与特定业务。关键的区别在于，在你做出这些定义之前，现实系统（我的意思是需要你开发这个系统的客户、办公室成员或部门）并不会向你提出这三个概念；除非你主动提及，否则这些概念也不会对现实系统的实务有任何影响；除非你将这些概念独立出来，否则即便现实系统的确是由这些规律内在地驱动着的，也不会有人发现。

但是，是何种思维方式，让你：从现实系统中“发现”这三项知识，并将它们设定为这样的一些概念，并为这些概念设定了有别于其他的依据？

又或者问：你何以在系统中做出一些设定，而非仅仅陈述现实系统的事实？

五

了解系统的一些具体方法，大体来说类似于图 4-5 所示的一个认知过程的方法树。

图 4-5　认知过程的方法树

如你所见，我们事实上只讨论了认知行为中很小的一个部分(5)。“识别”与“分别”是这个树上较低层次的方法，它们能得到系统知识而无法归纳之，能分辨出差异而无法梳理之，能构建功能模块而无法推演之。因为归纳（概念）、梳理（关系）、推演（逻辑）这些架构活动所需要的，都是较高层次上的思维方法。

现实中，基于所面对的计算机系统，我们大多数的系统抽象与建模过程中都会用到“分别”这一认知方法。比如说，我们将已知需求规划为条目，然后分门别类，进而整理出子系统、模块、服务，以及规划出服务器、集群等的方案。对系统中的组成、要件、关系等加以分别，是上述这些活动的基点。

而这只是系统的一部分。如果我们能据此“架构”出系统，那只能庆幸：这个系统在绝大多数情况下表现为一个数字系统，因而如前所述——是可以基于“数的值”这一抽象概念来进行“分别”的。

或者反之，我们无法架构出系统，因为我们无法通过这种方法来构建系统的知识。

第二节　知识的构建

一

识别与分别对于了解事物的内在特性来说，都只是辅助手段。而这就是能够建立一个系统的物理模型（组成/结构模型），而难于建立它的逻辑模型的根本原因。

因而我们需要关注在更高层级下建立知识的方式。事实上，我们在架构活动中进行的归纳（概念）、梳理（关系）、推演（逻辑），这些活动的核心基础在于图 4-6 中的“知得”而非“识得”。

图 4-6　认知过程的方法树：建立知识的两种方法

这里存在两个方面的、前设性的问题：其一，我们是否有能力得到一个物理模型；其二，我们得到上述物理模型的过程是否仅仅依赖“识得”。然而，这两个问题的答案都是否定的。首先，我们可能得到很多种物理模型，这些模型映射了现实系统的不同视角。真正的原因是：你难于一以贯之地采用特定视角去观察现实系统，并且你所了解的系统也会动态地以种种角度呈现给你。仍以上面的办公系统为例，通过一段时间接触之后，你会发现“总经理”这个角色的日常工作很难了解清楚。

□　一方面总经理很忙，他只能只言片语地向你介绍他的具体工作，因此你可能需要找到他的秘书来协助整理这些需求。而一旦主角由“他”变成了“他的秘书”，活动由主动介绍变成了侧面观察，“总经理在办公系统中的需求”就变得并不那么可信了。

□　另一方面办公系统本身就存在多面性，例如它既是每个人的私人工作平台，又是多个人

的公共协作平台，还是系统资源的承载平台……那么你观察这个系统的时候，得到的信息将会是向多个方向发散的，以至于你会觉得：向任何一个方向“多做一些”都会变成一个巨大的分支。

这些观察之间是冲突的、不相容的、重叠的、共生的等类似于这样的关系。在现实中，它们大多数时候都很融洽地、交织地存在，因为现实中的系统总是自洽的——系统中的角色总是在制造冲突的同时消弭着冲突，这就是所谓的生态，亦或“它们”之所以表现为一个（活着的、动态的）系统的内在能量。

但计算机系统只能描述其中的一部分（事实上这也意味着计算机系统只能解决动态的现实系统中的部分问题），这一部分必须首先成为“我们”——作为观察者的认识，而后才会表达为软硬件系统中的“可计算的”映像。最终，我们事实是通过对“映像”的运算，来还原现实系统中的某些侧面，以达到我们的目的——替代之、推演之，并作为其他可计算系统的组成部分。

所以事实上我们可能得到多个物理模型，它们在表面上看起来都是现实系统的“正确映像”，但其内在是各自不同的。例如，换个人来表达上面的办公系统（的物理）模型，可能就会是图 4-7 所示的样子。

图 4-7　通用办公系统架构 v0.0.0.2

二

这个“架构 v0.0.0.2”版本显然也是对现实中的办公环境的正确描述，但却与此前的“架构 v0.0.0.1”迥然不同。那究竟是什么原因造就了这样的差异呢？这仅仅是图形的组合方式的不同，还是某些“架构师”的个人喜好的差异，亦或是现实系统在本质上就存在着这样的多样性？

都是，但也都不是(6)。事实上当我们试图去表达现实系统的“一个映像”时，我们总是存有特定的意图。这种“架构上的意图”决定了我们的观察视角，也决定了我们之后表达的结果(7)。

“架构 v0.0.0.2”中包含了此前版本中的全部“业务”，并且认为：

□　业务是一个未知规模的“业务集”中的一部分；

□　业务之间是否存在“公共业务”与“特定业务”等分别，是不确定的；

□　业务仅仅是功能性的系统模块，与特定的使用者（用户）是无关的。

它还有一个“角色集”：

□ 角色集包含了此前版本的“一般用户”与“特定用户”等；

□ 角色集加入对系统持续观察后发现的一些新角色；

□ 角色集的规模是未知的，它可能随着现实系统的进化而扩展，也可能在某个（阶段交付的）计算系统中被确定。

最后，它还加入了一个在此前系统中并不存在的“组织机构”：

□ 组织机构是对现实系统的“组织”的重现，组织是一群有相互授权关系的人(8)；

□ 组织机构只表达了“人与人”或“系统与人”之间的授权关系，即“角色”；

□ 没有确定“一个人”是否能“被授权”为多个角色。

尽管“架构 v0.0.0.2”未能描述许多细节，例如是否交叉授权、组织本身是否有层级关系、业务之间是否有逻辑关系等，但是它准确地体现了架构者的一种意图：通过映射现实中的管理责权关系，而不是（如“架构 v0.0.0.1”那样）通过区别功能模块的适用群体来规划系统。

换言之，“架构 v0.0.0.2”体现的架构方向应该被称为办公管理系统，而“架构 v0.0.0.1”完成的则是办公（功能）系统。关键在于，前者体现了一种架构意图，而后者仅仅是对现实系统的一些事实的复制。然而一旦架构思想中出现了这一意图，我们就不得不提出如下的设问：

□ “管理”是现实系统的需求吗？

□ “组织机构”能够正确地映射现实系统的“管理行为”吗？

三

真实的情况通常是这样的：客户提出“办公系统”时，并没有打算开发这样一个寄予了管理期望的软件产品。从客户的角度上来说，这个软件的底线是帮他们减少一些手头的工作，并尽量让现行的工作更规范一些。换言之，客户在最低限度上需要的是一个现实的复制品与流水线。

通过现实系统的直接需求是推断不出“管理”这一概念的产生的。但是回溯我们此前列举的几点事实，其中：

□ 这一系统总是某些办公室成员使用的

是一个关键事实。这一事实模糊了“办公室成员”的类型。我们从两个方面重新考虑一下：如果这是某一个特定类型的办公室成员使用的系统，那么它适宜实现为一个工作系统，用来重现某种特定工作的规则与流程；如果这是一个混合的、由不同成员及其工作需求交织而成的系统，那么这个系统（的本身）必然需要某种东西来使自身规则化。

也就是说，“管理”不是现实系统的意图，而是映射这一系统到计算环境时的一个需求。我们必须确定：如果这一需求来自于现实系统，那么它是原始需求；如果它来自于上述的这个软件系统本身，那么它首先是设计者的意图，其次才是对现实系统的反映。

这是一个典型的因果问题：究竟是现实产生了意图，还是先有了意图再去参考现实。我们强调这一细节的原因于：如果是前者，那么控制这一意图（以这里的例子来说，是指“管理”这一行为）的意义在于“控制原始需求”；如果是后者，那么控制它的意义在于“控制设计欲望”。

一旦我们确认这只是一个意图，并且这一意图的核心仅仅是“规则化”那些需求与需求的用户对象，我们就需要更深层次地设定“被规则化的”这个系统（本身）。总结我对这一设定的考虑，它将会是：

□　与现实系统看起来类似的

□　具有同等的组织容量的

□　基本符合现实系统的运作逻辑的

一个软件系统。

这三项设定仍然都是架构意图。确切地说，这三项意图都是为了控制“管理”这一意图的规模的。从思考行为（的模式）方面来看，上述概念或观点的层进关系如图 4-8 所示(9)。

图 4-8　基于对思考行为（的模式）的观察：上述概念或观点的层进关系

与上述的整个过程类似，我们可以：

□　在“与现实系统看起来类似”这一方向上，发现类似于“经营”、“营销”、“人力”等这样的一些角色，并进而形成“角色集”；

□　在与“基本符合现实系统的运作逻辑”这一方向上，发现“经营”角色管理“营销”角色的市场方向，“人力”角色提供“营销”角色所需的资源，但并不管理之，等等逻辑；

□　在“同等的组织容量”这一方向上，发现这是一个具有“规模不会超过 200 人的”、“不需要跨国管理”等限定条件的系统。

通过对种种方向的探索、思考与推演，我们会得到更多的、类似上述的这些设定——设计原则或系统原则。需要留意的是，现实系统中，并没有任何需求方来提出这些设定，例如经营角色会说“我们需要一个饼状图”，营销角色会说“我们每周至少发布一次营销活动”，但是他们都不会说“你的系统中需要这样有着管理层次关系的两个角色”。

回溯上述过程：管理、角色集、组织机构这三个概念的提出，都是架构师的架构意图，以及基于这些意图进行推演的结果。

四

意图，是“识得”的核心，即“你想要什么”决定了系统如何构画，而不仅仅是对现实系统的复制。

意图是架构真正的灵魂。架构活动只是将这种意图表达在架构产出中，并阐述这一意图的合理性；如何得到或形成意图才是架构的精髓，其本质是通过抽象过程，对既有系统的再认识与再创造。简单地说，如果架构师没有意图，那么系统只是目标系统的某一时间上的静态映像(10)；而架构师如果有意图，那么系统也就有了灵魂，就能跟随目标系统的实际需求的发展而演化，或至少为这种演化留备了可能。

“管理”是为了“应对不同类型的办公室成员”而引入到系统中的概念，而“组织机构”则是对现实系统的“管理行为”的一种映射。我们并没有要求——事实上也很难做到——用软件系统来映射现实中的全部管理行为，譬如尽管我们可以通过公文流转去映射“总经理向营销经理下达了业务指标”，也能够通过授权变更去映射“人事任命”，但是却很难设计一个东西来反映“某员工因业绩受到了表扬而积极性大增”——尽管这也是现实中的一个管理活动。

但究竟是什么决定了我们选择“组织机构”而不是其他的东西去映射现实行为呢？

我认为这存在两个条件：其一，它首先必须是能够被规则化的(11)，这是主要条件；其二，作为附加收益，规则化也可以为其他系统构件带来便利。只有在主要条件成立，并且附加收益丰厚的时候，我们才会确认这一“意图”的必要性。

对这两个条件的思考是架构过程中的一种权衡，即从“想要什么”到“能要什么”的一个过渡。这个过程中，“控制架构欲望”是一种关键素质。而这一素质的源起与核心，是架构师对自身职责的不断的、反复地省思，即“想要什么”应当能决定一个系统在时间与空间两个方面的特性，而不是（仅仅）出于客户需求或自我喜好。

组织机构表达的“人与人的授权”(12)以及“被授权者是可以行使系统行为的角色”(13)这两点是可以被规则化的。这很明显。但是它能带来哪些便利呢？表 4-2 是一个简要的考察表。

表 4-2　面向架构话题的考察：授权与角色规则化带来的主要收益

① 对于“角色集”及其权限来说，可移植性的主要方向为抽取独立的数据层，表达为格式文本或数据库。对于系统的其他功能或逻辑部分，“可移植性”涉及跨平台、跨系统以及交互设备兼容性等多方面的因素。

在这些考量中，最重要的是“概念完整性”，它决定了整个系统的核心逻辑，以及描述架构、功能与内部关系的一般方法。如果一个架构设定没有概念完整性方面的必要，通常它的价值收益就会偏小，偏局部，或者可备选。

最后需要补充的是：这样完备地考察通常是不必要的。这是因为，其一，很少有类似授权这样的架构意图，是能够影响到系统全局并在各考察点上都有相对平衡的重要性的；其二，架构意图通常是反向论证的，即“它不与哪些考察点冲突”；其三，核心架构意图通常是明显的、一贯的以及关键的，因此它的影响面也就巨大，这意味着多个这样的意图并存时将是轻重缓急的问题，而并不是是非取舍问题；其四，如上的轻重缓急是一时的选择，可能会随着所架构的系统——或者说项目——的推进而有变化，这既说明了多种意图的必然性，也说明了多种意图间冲突的根源，亦即是需求的内容与焦点会随时间与空间变化。

最关键的架构意图是架构师对上述第四个因素的推定，而并非依赖当前的、静止的需求。这种推定的合理性是建立在一个非常完整、缜密、基于抽象概念的逻辑推理基础上的，其背景多数已经超出了“软件系统”本身。例如，对于办公管理系统而言，一个推理的基础是“将更多的管理功能置于办公系统是一种趋势”，这一推断可能来自于：

□ 跨地域的办公环境中，网络办公系统是一种成本更低的选择；

□ 更大规模的公司中，办公资源的管理是人力难为的；

□ 流程化是复杂的公司组织中减少错误的必需；

□ 文档的管理可能失控；

□ ……

而最终在这些信息的基础上推理逻辑可能是：

□ 成本控制指数=企业规模（电子化投入增幅/管理投入增幅）

只要在相对应的企业规模下，成本控制指数小于 1，则企业将必然“乐意于”将更多的管理功能置于办公系统。

五

“知得”是一个由抽象概念开始的思考过程。在我们的架构活动中，我强调这是一个由“架构意图”驱动的抽象活动。但这并非惟只的方向，并且可能是一个本末倒置的方向。这里需要强调两点，一是我们并没有完整地讨论“架构意图”的由来(14)，二是“本末倒置”并非是一件坏事。

我们要实现的系统只是现实系统的一个映像，这是我一再强调的。这意味着它与现实系统近似而又不完全相同。近似表明它与现实系统的关联，这种关联来自于对现实的观察、分析以及由此进行的知识构建活动；不完全相同，表明它与现实的差异，这种差异来自于架构师对

系统的设定，亦或者说架构师强加于系统之上的架构意图。

从经典的架构与设计的法则来看，是“需求决定架构及设计”，这种需求通常是以现实系统为核心的。这很合理，毕竟从上述的分析来看，现实系统才是系统的本体，系统只是现实系统的一个侧相，而“架构意图”只不过是由架构师对系统之所用的理解。我们一旦强调由所用来推动架构过程，而忽略了本体的真实与侧相的含义，那么往往就会被指为本末倒置。

但是首先，对本体的认识方式决定了我们不能基于它来建立系统。本体不是观察而得的，也就是说，即使是用知得与识得构建的知识，仍然只是表达本体的一个侧相，而非本体的自身。这就是不同的架构师去观察与构建同一个系统的结果并不相同的原因——无论他们如何关注本体，如何尽心竭力地去阐述它，并证明自己的阐述是惟只正确的观察。本体是通过建立感受而形成的，如果一个人对现实系统不能设身处地，无有感同身受，那么他构建的系统会离现实系统很远。但是，如果这一观点成立，那么对本体的感受事实上是不可叙述的，任何语言文字叙述的“这一系统”都立即成为一个侧相，因为叙述可以有万千种渠道，万千种方式或万千种程度与细节上的差异。所以本体是唯只的，感受的结果也是唯只的，但到了表达为侧相，却不唯只了，是所谓“只可意会，不可言传”。

其次，本体的复杂性决定了我们不能基于对它的感受来建立这个系统。佛陀的拈花一笑是无解的：相对于其真实的本相来说，任何参悟者从这一公案(15)中所得的寓义都是确实的，而将它应用于任何思维活动的解说都是可行的。真正的原因并不在于“拈花一笑”这个行为外在的、形式上的简单，而在于“佛陀”这一背景的丰富。源于背景无以穷尽的丰富，对于任何其外在观察的解说都将趋于合理；对于其任何内在的感受都必将是、也仅只是对真实的无限逼近。本体的复杂性并不来自于一个形式或表面，而是其背景与历史的全体构成：全部影像以及影像的关系的集合。若基于如此复杂的本体来建立系统，其结果只会是三个字，是谓“不可说”。

我们需要反思此前系统(16)中所出现的概念：管理、组织机构、授权、角色（集）、业务（集）、日常办公、电子化管理，以及特定功能。在这里，我们并不讨论它们的确实含义，也不讨论这些抽象是否必要以及是否足够纯粹。我关注于一个实际问题：它们因何而来，如何来，以及为何不会成为你思维中的一个闪念并随之消逝而去。

我们需要反思这些概念在架构思维方法中的意义。

所有的这些问题都指向一个答案：架构意图。我们可以找到这样两个不同的架构：它映射同一系统，由不同的架构师来实现。当我们对这样两个架构作分析时，一定可以找到一些相同的部分。这些内容大体来自于由需求驱动的架构方法，它们是架构师对需求的正确描述、复制与映射。如果这些描述、复制与映射中存在了差异，在这两个或更多的架构师之间是可以调和的，因为这仅仅是对一些真实可见、可以反复而又唯只陈述的需求的不同看法，它可以论证、分解、削弱或搁置，无论如何，它们不会成为两个架构中最核心与典型的差别。

我们也必然会找到一些不同的部分。（除了上述的差别之外，）不同的部分必然来自于架构意图的差别，是明显的主观认识，带有很强的目的性。举例来说，如果架构师 A 将系统设计为基于数据流的，可能的原因是：

□ 他熟悉一门数据流语言；

□ 他的团队有过此类开发经验，熟悉这种架构；

□ 他特别关注于系统中的数据活动；

□ 他掌握某种成熟的数据流架构实现；

□ 他有过数据流架构的成功经验；

□ ……

如果架构师 B 要理解这一意图，除了在知识积累上要与 A 类似之外，还要对 A 的上述背景也有所了解。换言之，事实上架构师 B 是要了解“目标现实系统+将架构师 A 作为系统对象”这整个的全集。对于架构师 B 来说，这是很不公平并且难于实现的。所以，往往架构师 A 无法期望别人理解，而只能寄期望于别人接受他的架构意图。

但是，如上所讨论的，如果它仅仅是意图——既有主观性、目的性，也有强制性，那么就可能是某个个体或利益干系人的一己之私。例如，某个架构师在会议中所发之言论，可能只是向其他架构师的权威挑战，而并非是系统在架构方面的真实所需。因此，如果我们不确切地定义“架构意图”，那么从种种意图中找出真实有意义的架构意图这一活动，就会变成一种类似修炼的东西，进而变得无有方向、无有所指，因而也无法辨识。

六

那么，架构意图到底是什么呢？

架构意图需承架构的定义而来，它首先必是“经营角色对方向的设定”在系统上的体现。若架构意图不体现方向，则它将只是局部的、边角的一些架构决策(17)或意图(18)。架构师的核心价值，在于通过架构意图来将“方向设定”映射为“规模与细节”。其中，“规模”表现为架构的边界/范围，“细节”表现为架构部件的联接关系/联接件。

对于架构意图的识别，有三个入手的角度。这三个角度仍是“规模与细节”相关的，其一，是系统的脉络；其二，是系统的组织(19)；其三，是系统组织间的关系。如果一个意图表现了架构师对系统上述三个方面的理解，则该意图应当视为架构意图(20)。

第一个方面，系统的脉络是对方向性的体现。这包括系统内在的动律与整体的动向两个方面。前者，即内在动律意味着架构师应当对系统（作为一个整体）的核心运作规律加以考察，这包括一般过程、限制条件以及最基本的系统要素间的流转关系。下面以支付系统为例。

□ 一般过程：在某种支付场景下，用户 A 与用户 B 之间的一次资金转移。

□ 限制条件：支付场景、用户 A、用户 B、资金以及资金的转移这五个因素是否被“当前系统”所理解。如果不理解支付场景，则应该将上述过程实现为流水，反之可以实现为一笔

交易或基于订单的支付过程；如果不理解用户 A 或用户 B，则应该将上述过程置于一个会话或在过程中使用 token/ticket 以识别(21)，反之支付过程应当自行决定是否需要对用户进行复核（例如站内消息或手机短信通知与验证）；如果当前系统不理解资金转移，则应当记录支付行为并提供外部查询接口，反之则可以完成资金转移。

□ 流转关系(22)：其一，用户 A 与用户 B 之间的消息通信（可选）；其二，系统与用户 A、用户 B 的消息通信，例如短信确认与通知等；其三，用户 A 与用户 B 的资金账户中的资金变化。

后者，即整体动向意味着架构师应当对系统的长期目标做出考量，这包括(23)：系统是独立系统还是公开系统，是规模渐增还是功能渐增的系统，是战略上的布局还是战术上的一个实现点。以金融业务中的清算系统为例(24)，它通常是一个独立系统，因而不需要公共接口；它会随业务量的增加而规模渐增，但不会在系统功能上有明显变化；它通常只是一个关键技术点，而不会影响到整个金融业务的战略构画。

第二个方面，系统的组织是对范围的考虑，它主要讨论将哪些内容放在一起的问题，它一方面决定组织在内含上的规模，另一方面也决定组织在外延间的距离。以上面的支付系统为例，系统中是否完整包括“支付场景、用户 A、用户 B、资金、资金的转移”这五个构件，是需要被确定的。正如上述——在第一个方面中的分析所体现的，系统的组织既决定了“限制条件”的细节，其本身也取决于对系统脉络的分析。亦或说，很难孤立地看待系统脉络与系统组织，它们是在一个完整的、整体性的思考过程中的反复权衡。这一权衡的基本依据是上述的整体动向，例如若支付系统开放(25)，则账户 A 和账户 B 必然要从支付过程中抽取出去，并且相关的流转关系必然依赖外部系统；若将支付过程理解为功能渐增的系统，则它必然不适宜开放，因为这意味着接口趋向于应付功能变化，进而导致接口变化——而接口频繁变化是不合理的；若它本身只是战术实现点，那么它的开放基础就将建立在技术方案（如数据架构或系统群集等）之上，难以对行业、渠道或领域构成实质性的影响。

第三个方面，组织的关系是对联接件的考虑，它主要讨论上述组织成员间的通信，并进一步决定通信的形式与其成本(26)。仍以上面的支付系统为例，我们假定(27)用户 A、用户 B 与支付场景都是外部系统，那么我们必须考虑的联接关系就包括：其一，用户 A 和用户 B 是否具有消息通信过程，如果有，应当如何实现，例如是实现为专用网络中的通信客户端，还是使用类似手机短信这样的第三方通信网络；其二，用户 A 和用户 B 与支付场景是否有关系，是否在支付场景中通信，例如站内短信、通知等；其三，这些外部系统与（当前）支付系统之间是否有通信关系，例如安装服务端通信模块，或依赖于在外部会话中建立凭据。

架构意图中最重要的是系统的脉络，其整体动向是本质性的需求，其内在动律是上述需求的表现与表达方式(28)。总体来说，任何一个架构意图的形成都是对三个“入手角度”整体的反复考量进而形成的一个最终认识，而决非其单一方面的阐述。例如，以上述的支付系统来说，最终的架构意图应叙述为(29)：

一个跨领域的开放支付平台。

第三节　最初的事实

一

三人成虎这个故事，出自《韩非子》：

魏国大臣庞恭问魏王说：“现在有一人来说街市上出现了老虎，大王相信吗？”魏王道：“我不相信。”庞恭说：“如果有第二个人说街市上出现了老虎，大王相信吗？”魏王道：“我有些将信将疑了。”庞恭又说：“如果有第三个人说街市上出现了老虎，大王相信吗？”魏王道：“我当然会相信。”

这则故事的有趣之处在于：所谓“事实”的形成，与事实本身看起来没什么关系，而仅仅在于观察者的主观判断。这看起来相当地可笑：“科学”总是依赖客观事实，但我们对客观事实的认识本身就是发自主观的——从思维的角度上来讲，如果“毫无主观判断”，那么我们也就连任何概念(30)都无法形成。

到底什么才是事实？如何确认我们认为的事实就是事实本身？若我们孤立地看待“形成概念”这一过程，就会陷入上述的吊诡。真正完整而科学的思维方法是将“概念、论证、应用”三者合而为一的：单独地提出概念确实是主观的，科学之谓科学，在于通过后面的两种行为使概念符合逻辑论证与现实实证。

我们此前基于三点事实来讨论了“架构 v0.0.0.2”，我们称这些事实“显而易见，是由系统本身决定的”：

（1）这一系统总是某些办公室成员使用的；

（2）这一系统总是提供上述人员的日常工作所需的功能；

（3）这一系统既包括对现实工作的映射，也包括一些试图改变现行工作的电子化需求。

但真相未必如此。如此前所述，在这些“事实”中，我们必须认清哪些是对客观事实的叙述，而哪些是主观判断。

置疑那些主观判断，是系统架构思维中的第一步。

二

我们谈到过系统的脉络在架构意图中最为重要，它包括两个部分，即内在动律与整体动向。通常，前者是无可争辩的事实，后者则是主观判断或客户战略。

考察上述“三点事实”，其中只有第一点是“办公系统”的事实，但它构成不了内在动律，

因为内在动律应当是对一般过程的描述。只有将与第一点和第二点事实合而为一的时候(31)，才能表达如下的一般过程(32)。

□ 一般过程：办公系统向某些办公室成员提供日常工作所需的功能。

接下来我们讨论事实之三，亦即是：这一系统既包括对现实工作的映射，也包括一些试图改变现行工作的电子化需求。其中，“系统包括对现实工作的映射”是完全正确但又毫无意义的——这是软件系统的本质含义，大体上来说是放之四海皆准的。而后半句就不见得正确了，因为“试图改变现行工作”并不见得是事实。

是否需要通过办公系统来改变现行工作，是客户的一项决策。如果在客户对业务的电子化战略中是有这样的需求的，那么这可以先“暂且”作为一项事实放在这里(33)；如果没有这样的需求，那么它就是一种相当危险的臆断。

先讨论它的危险性。“改变现行工作”将会涉及的问题是业务流程再造（BPR，Business Process Reengineering），它直接将“办公工具”这一简单的解决方案推进到了“企业流程化”这一系统的规划工程中。它涉及目标企业是否有能力展开 ERP（企业资源计划，Enterprise Resource Planning）过程的问题，涉及该企业对其资产背景、管理模式、行业领域以及长期决策能力的评估。因此，是否“改变现行工作”是架构师无法直接从上述的一般过程，亦即是从第一点和第二点事实中推断出来的。

架构不是为客户设定战略(34)，而是服从客户设定的战略。如果客户没有形成战略，那么架构也就只能依据内在动律来主观判断整体动向。如前所论，这种主观判断是依赖对系统的分析而非对战略的假定的，并且也主要用于描述系统的发展方向，而非系统的客户——企业或领域的发展方向。

所以在架构意图上中对整体动向的考虑主要是三点：依赖、复杂性与持续价值。这三方面的问题都可以从战略设定中去寻求最终答案，但如果战略不清晰，则也可以从上述的一般过程中去得到一些（阶段性的、可维护系统自身的发展所需的）设定。例如，在“架构 v0.0.0.2”中引入的管理这一概念（以及架构意图），其实是对系统的长期目标作出的考量。

□ 整体动向：提供面向办公室成员和所需的办公功能的管理功能。

进一步地，对于整体动向的三个方面的问题的思考可能是：

□ 是企业内部的独立系统(35)；

□ 是功能渐增的系统；

□ 是战术过程中的一个实现点。

看起来，我们最初设定的“三项事实”许多是作不得准的。但这并不影响我们进一步架构该系统。与其他工程活动一样，架构工作也是渐进的，并不是一开始就准确无误。因而架构思维中需要不断反思与回顾，而架构活动也将是持续与迭代的。

在现阶段，伴随着“架构 v0.0.0.2”而来的，是我们确定了一项事实与一个架构意图。这是我们目前能做到的、有关“系统架构”的全部思考。

三

我们的两点设定带来了一个“没有战略”的死结：

□ 客户并不清楚战略；

□ 架构是服从客户设定的战略，而非为客户设定战略。

因此“依据内在动律来主观判断整体动向”其实是不得已而为之的，因而我们也必须质疑：作为架构师，是否真的“不能设定战略”呢？

战略与方向，是存在本质性的不同的。方向只表达动向，而战略其实是已经决策的动作，或对其行动步骤的规划(36)。对于架构师来说，无法决定的其实是客户的战略决策，但对于客户的方向是可以有自己的判断的。以某一企业客户为例，架构师可能不能决定该客户：

□ 是否通过购并消灭 30%的竞争公司。

但架构师可以考虑：

□ 购并是企业客户的一个可选途径，若该途径成立，则账户合并将会成为系统中长期的关键问题。

我们看到，这里存在架构师必须“考虑到”的两个因素：

□ 其一，消灭竞争者是公司运作目的；

□ 其二，购并是达成上述目的的一个可行手段。

架构师对“方向”的考虑在于上述的第一个因素，即对“公司运作目的”的判断：某些情况下，公司并不存有这样的目的，或公司可能根本没有短期竞争者；某些情况下，公司可能需要跟竞争者在某些领域合作，而在某些领域相互牵制；某些情况下，消灭竞争者只是公司经营者的一个口号，而非阶段性的目标……对于类似这些问题，是架构师在方向问题上必须有的判断，这些判断才是后续“是否需要将账户系统的通用化作为架构意图(37)”的依据。

但无论是否以此作为架构意图，架构师在整个过程中都没有将“购并”作为一个客户战略。架构师只是在客户的经营动向上作出了自己的判断，更或者是尝试性地思考了这一方向的可能性，并更进一步思考了架构需要为此而做出的准备。

是否将“通用账户系统”作为架构意图，是依赖对许多假设条件的分析而作出的，决非凭籍上述一个推断而得。需要指出的是，上述的思考背景包括许多方面，例如：

□ 架构师是否有客户领域的工作经验；

□ 架构师是否对相关行业的信息有过统计分析与评估；

□ 架构师是否了解客户或其领域当前面临的主要问题；

□ 架构师是否参与客户的经营决策；

□ 架构师所面临的系统是客户的解决方案还是试探性产品；

□ 架构师所面临的是对客户系统的改造还是重建（或新建）；

□ 架构师是否了解客户对该系统的持续投入情况。

对于这些问题，一方面它并不仅仅取决于系统内部的一般过程，另一方面它也不是对客户战略的直接设定。它们（以及更多的问题）综合地反映了在架构问题上的、最后可以依赖的一些信息，亦即是：客户在系统——所对应的现实系统——中可能选择的方向。

这种情况下(38)，架构师需要一些主观判断的能力，以及在系统推进中的一些尝试机会。

四

可见，在既不存在所谓事实（因而也难有可信的主观判断），又没有所谓战略时，我们是可以藉由前瞻方向来形成架构意图的。而另一方面，我们也可以尝试回溯问题。

Soul 曾经是我邻座的同事。某天一早，他敲响了我的工位的隔板，问道：Hi，Aiming，你有没有一个苹果？显然我不会一大早地备好了苹果来等着他的发问。因而我只是诚实地回答道：没有啊。然后机械地开始一天的工作：打开电脑、泡茶、拿出文件夹，以及把椅子调到合适的角度……

等等，我们在谈什么(39)？其实，仅仅在几十秒之后，我便醒悟了——Soul 的“一个苹果”不过是一个需求。我有或没有一个苹果，或者能否帮他找到一个苹果，或者他找到一个苹果之后是否太酸太甜等等细节，都只是由这个需求开始的求解（又或者面对这个需求时，就如同我当时的茫然无解）。一刹那之间，我便绕过了这一需求的纷繁枝节，再次回问他道：你是不是饿了啊？

这个场景之所以让我如此印象深刻，是因为在前半段中我经历的是典型的“程序过程式”的思维活动，如同图 4-9 所展示的。

图 4-9 “程序过程式”的思维活动

我们的软件开发活动向来是从对需求的分析开始的，经过设计、开发等过程，最后交付和维护。这一过程是如此的自然，因而我们将它从历史的开发活动中“识别”出来之后，立即被看成是软件工程作为一个成熟概念的标志(40)。然而在上述场景的后半段，我并没有立即陷入对“（能或不能，以及具体如何）满足需求”的挣扎之中，我从 Soul 的需求中回溯到整个系统所面临的问题：“要一个苹果”是因为饥饿吗？(41)

如果真实的问题是“在这个早晨，Soul 饿了”，那么饼干、馒头甚至一杯早茶都可以解决他的问题。为什么一定要去满足“一个苹果”的需求呢？可见，我在这后半段中所经历的，其实是图 4-10 所示的另外一种思维模式。亦即是说：我们可以暂时将“满足系统需求的急迫性”放在一旁，而去寻求“究竟是什么问题导致了这些需求”。当我们看到了问题，也就同时看到了解的抽象。

图 4-10　面向问题的思维活动

煮一枚鸡蛋真的是一个需求吗？当我们把煮鸡蛋视为一个系统的时候，一切前设都既成定局：我们只关注于一枚鸡蛋的生熟与安全，而并不关注“究竟是什么”导致了我们要去煮一枚鸡蛋。同样的问题被放大无数倍之后，我们也只是关心一锅鸡蛋与一枚鸡蛋的煮法的异同。我们往往在做一些具体行为的时候，忘掉了它的源起，这便如同《大道至简》中谈到的愚公一家，数百年的移山工程结束之后却不得不去筑关自守。

需求不是问题，但需求是问题的表现。例如，Soul 的问题是饥饿，而表现却是需要一个苹果。我们所关注的系统大致也是如此，它可能表现为一系列直接的需求，因此我们可以从需求中找到事实或直接获得战略设定；它也可能表现为一些看似无关的间接需求或间接问题，在这些表面现象的背后，总会有一些核心的问题是它们的真实动因。

“真实动因”是一个关于“找到问题”的极佳设问。通常，我们对需求的描述都是“谁，做什么”，接下来是一些限制条件，例如“在哪里做，如何做”(42)——这是我们一般性的需求分析方法。而在这些行为中，提出“原因”这个疑问，就是为这一需求找到它的动因。当我们的多个需求都指向相同的、相类似的动因时，我们就已逼近核心问题——我们最终缺乏的，仅仅是将这些动因归纳成问题而已。

什么是问题？问题有两个形态：其一，若系统存在某种“一般过程”，则阻碍这个一般过程的，必然是核心问题；其二，若系统存在一个确定的观察者，则所谓问题，就是这个观察者的期望与现实之间的差异(43)。换作精炼一点的描述：

所谓问题，要么是系统与其要素之间的矛盾，要么是观察与其预期之间的矛盾。

我们在系统中引入了一般过程与观察者，前者是“导致问题”的内因，后者则是其外因。系统的不变性，一般来说是由前者决定的，所谓平衡，即是在这个一般过程的要素之间的、时间与空间上的权衡；系统的变化往往是后者导致的，亦即观察者——例如经营角色或主管——对于系统的期望缺乏一贯性。

五

在此前的讨论中，除了对基本的事实与真相的识别之外，大概涉及三种思维方法：设定、试探和归纳。而本质上来说，这三种思维的过程也是模型与概念抽取的过程。由此所得的，是一个可以用来作为基础并进一步讨论的现实系统的映像：软件系统的架构。

但这个映像不是解，而是讨论解的一个工具。

我们讨论过“路人甲过河”的问题。我们说，在路人甲过河之后留下了“船”，如果能复制“船和行船的方法”这些知识，那么我们可以用相同的方法过河。

但是，有一些问题被这一过程给掩盖了。其一，甲如何造船？其二，甲在面临问题时，是如何得到“造船”这个解的？第一个问题中所包含的知识可能已经佚失了——当我们已然面临“一艘船”这个事物时，这是相当有可能的；而第二个问题中的知识则是不可复制的。

大多数情况下，我们并不去追究这两个问题。这是因为当有一艘船在我们面前时，这两个问题就显得与“过河”无关了。而这也是典型的程序员思维(44)：关注工具之用。

架构师所面临的往往是“河”这个事实，以及“过河”这个问题。架构师的思维应当是基于对事实与问题的思考，而非基于可能既有的、已得的“船”(45)的应用与设问。

有了船，我们的衣服不会再湿；但如果我们愿意湿掉衣服，我们仍然可以过得河去。

河其实未必很深。

知道“河其实未必很深”，是大智慧。

————————————————————————

(1) 当你能站在客户的角度去感受他所知的系统时，你能了解到他的所需。这何尝不是需求分析的一种形式？

(2) 这在中国古代哲学中称为“名实问题”，即名实不知、知名不知实、知实不知名，以及名实如一的问题。

(3) 不一定是指“触摸”，可能是看见、听见、嗅见等多种接触形式。不过，我并不确知“第六类接触”是否算作其一。

(4) 事实上并不尽然，我只是有意地忽视了这一过程的复杂性。

(5) 这个认知树仍然是整个认知体系的一个局部，并且也绝非表现为这样规整的二元划分。但是限于我的能力以及本书的主题，我无法讨论更多的内容。

(6) 做第一种回答的人，是关注到了不同视角下的差异，因此会把系统变得更为复杂的。做第二种回答的人（亦如后文中的讨论），则试图进一步从差异中找到共性，从而简化对系统的讨论。

(7) 这并不一定决定我们的表达手法。的确存在这种可能：手法不同，但“表达的结果”中所呈现的意图却是相同的。

(8) 某些组织关系中，并非是单一的“授权”问题，因此这种表达方式也并非万能的。

(9) 此图用于反映思考与决策的过程，阅读图解的基本方法是：当我们将“管理”作为“架构意图”，进而决定了“与现实系统的相似度”，例如：与现实系统看起来类似。

(10) 这并非不可取。许多生产系统就只需要“再造”现实系统的现场即可，但这也并不表明架构师不需要“意图”，因为再造的过程本身也是动态的，也可能是阶段性的（例如一期、二期这样的分期工程）等。

(11) 仔细分析“规则化”本身，它可能是结构化的另一个代名词。在现实系统中的一部分行为可以被抽象为“逻辑的规则”，例如营销活动策略、折扣策略等等；另一部分则可以被抽象为上述行为“依赖的信息”，例如参与者信息、依赖资源的信息、事件信息等。这两类规则化行为与结构程序设计中的“数据+算法”有着抽象上的近似。

(12) 它决定了逻辑“if Auth.isManager(User1) then Auth.assign(User1, User2, 'aAuthDescription')”的合理性。

(13) 它决定了逻辑“if Auth.has(User2, 'aAuthDescription') then doSomething(…)”的合理性。

(14) 严格地说，“管理”这一架构意图的由来是我们在“第二节　知识的构建（三）”中讨论过的——它来自于基于现实系统的几点事实的推论。但是存有两个问题：其一，这一推论过程是经验化的；其二，我们未讨论有关“架构意图的由来”的思考方法，而只是陈述了它的某一个实例（即“管理”）的由来。

(15) “公案”，佛学禅宗用词，是对高僧言行的记录，可用作思考对象、座右铭，以启发思想，供人研究等。

(16) 目前来说，“架构 v0.0.0.2”是一个不错的架构。尽管它离投入开发实施还很远，但就我的经验来说，它在“正确地架构一个系统”这一方向上有着相当可喜的表现。

(17) 架构决策是下一章要讨论的问题，但许多时候架构决策被混用为我们这里讨论的架构意图。

(18) 这里的“意图”是指某些与架构无关的意图，或阶段目标的架构意图。

(19) 组织在这里是有两层含义的：其一，它作为名词以表明组成部件；其二，它作为动词

以表明部件之间的结构过程。

(20) 意图并不一定是“单纯”的。架构意图可能同时蕴含了架构师在其他方面（如公司政治、市场决策等）的考量，但我们这里只讨论其在外在表现上能否作为在系统架构与设计中一以贯之的架构意图，而忽视了其他方面。

(21) token 与 ticket 是常用的系统外认证的技术。这意味着具有某个认证系统专门来验证用户 A 或用户 B 的真实性，并在系统间将一个或一对识别用的凭据 token/ticket 传递给当前系统与用户，以使当前系统能够可靠地识别用户。

(22) 一个完整的支付行为涉及资金流与信息流，后者主要用于保障一个支付行为的可靠性。当考虑用户间的信息流转时，它可能是一个用户授信的支付系统；当考虑系统与用户的信息流转时，是在支付过程内加入了安全需求。

(23) 这里只是列举了三个设问，它们事实上分别反映的是系统的三个方面的特征：依赖、复杂性与持续价值。

(24) 这里讨论的是内部独立清算业务，跨行（银行间）与跨系统清算等业务与此有相当大的区别。

(25) 这里的“开放”是指将支付系统作为一组可公开调用的服务，提供给第三方公司或其他领域中的业务过程使用。

(26) 总体来说，我们是要削减通信成本的。因此，应尽量要求组织成员间无关系、不发生通信行为，或在发生通信行为时尽量不对当前系统的“一般过程”构成影响。例如，即使支付过程的确需要用户 A 和用户 B 发生一次通信来增强安全性，那么也应当尽量在支付场景中通信，而不是在资金转移中通信，亦即是将这一行为视为交易安全，而非资金安全。

(27) 该假定是以“需要实现为开放系统”这一动向为基础的。

(28) 也许不应该如此片面地定义整体动向与内在动律的关系，究竟是整体决定内在，亦或反之，是有哲学思考的意味以及观察角度的设定的。另外，质变与量变也是脉络中的一个关键思考，例如逻辑复杂性是否会决定架构意图？

(29) 架构意图的叙述是简洁且毋庸置疑的。例如，此前的办公系统，在“架构 v0.0.0.2”中的架构意图就是：它是一个管理系统，而非业务系统。

(30) 这里用“概念”一词包含抽象、观点、判断、认识等多种主观认知结果。

(31) 从语言严谨性上来说：第一点事实是正确的，而第二点事实并不完整，仅为第一点事实的补充。

(32) UML 中的用例图（use case diagram）是这种一般过程的形式化方法。但是，用例图是从用户视角来阐述与表达的，因此通常它将这一过程体现为（亦即是用例图表达的含义）：办

公室成员使用日常功能。

(33) 在现实的情况中，客户可能是盲目的。客户有自己的语境、目的与表达方式，因此将客户需求或叙述直接作为“事实”是相当可怖的。

(34) 如果真有这样的情况发生，那么应该看看架构师是否同时承担了战略决策者或顾问的角色。我事实上乐观地认为（系统的、整体的、面向全局的）架构师应当参与战略过程，但在这里只能谨慎地摒弃这些因素的影响。

(35) 如果用户整体的电子化程度相当高，则可能提供有限的、功能性的开放接口。

(36) 我们要尽可能避免望文生义地去理解这些概念。基本上来说，方向与战略并没有非此即彼的边界。事实上在一些讨论中，方向也是战略的一部分，例如将战略表达为战略方向、战略方案、战略决策等的统一。同理，架构师与决策者也并不具有那么明显的分别，很多时候架构师也是决策团队的成员。问题是，如果我们非得要分离出一个“架构师角色”来，那么我愿意将架构师作为战略的分析、细化和推进者，对决策过程只作辅助，而认为战略的制定者另有其人。

(37) 我们这里假设了一种情况，即在某个系统的建设中，架构师将“使用统一的、通用的账户”作为架构意图。而我们的讨论要点在于：是哪些因素决定了架构师应当（或不应当）作出该判断。

(38) 领域产品、客户项目、通用应用与自用系统，这些都因涉及的系统对象（及其用户、用户的认识）的不同，而导致对它们的架构过程不同。在这里讨论的主要是：当我们需要去提供一些领域产品，而对该领域又并不十分清晰的情况下，我们可以依赖哪些“信息”来构建事实并进一步地提出架构意图。这其实也可以应用于当前系统与现实系统由于高速发展而暂时性地失去方向的情况——它们都处于一种类似“三岔口”的局面中，需要重新的选择与定位。

(39) 就是这样日复一日，行走得如同一具木偶？木偶总是在“一如既往”地满足着需求，但它并不清楚这究竟是带来了提线人的快乐，亦或是观众的快乐，或者他们是否真的快乐。然而，这就是“行进于一个过程，而忽视目标”的工程活动的本相。

(30) 由 W. W. Royce 在 1970 年于论文《管理大型软件系统开发》中提出的、从需求开始的、基于“瀑布模型”的软件工程活动，他事实上是提供了软件开发的基本框架。

(41) 正确的问题是：要一个苹果的“原因”是什么。而“是因为饥饿吗”则是对该问题求解的一个设问。

(42) 参考 Harold D. Lasswell 在论文《社会传播的结构与功能》中提出的 5W 传播模式，以及基于此提出的 5W1H 思维方法，即原因（Why）、对象（What）、地点（Where）、时间（When）、人员（Who），以及方法（How）。

(43) 参考温伯格《你的灯亮着吗？》。

(44) 在《大道至简》中我将之称为“工匠思维”，那是更为确切和形象的。

(45) 仅以架构的实作而论，我们是有许多这样的船的，如三层架构、COM 架构、数据流架构等。

第十一章

架构是过程，而非结果

本章讨论的“系统”与上一章不同，这里的系统是一个规模用词，因此本章将基于“领域集”的概念来讨论系统问题。出于这一点设定，本章也提出了“架构师团队”的问题，进而提出了将“架构师团队面临的架构整体”作为一个系统（亦即是一个新的领域），来加以讨论，并且将这样的系统的架构目标称为“系统架构”。

就这一系统架构的构件而言，有两种可能的模型。第一种可以理解为架构组成论，它可以看成架构在空间视角上的求解，其部件应当包括：子系统、通信与验证。关于这三个构件的必然性，是基于几点简单的推论：若不存在子系统，则没有所谓“系统架构（之整体）”；若不存在通信，则子系统之于整体是无意义的，即它不必存在于该系统架构；若不存在验证，则子系统之于整体是不确定的，即它在系统架构中是或有或无的。第二种可以理解为架构形成论，它可以看成架构在时间视角的求解，其部件应当包括各种架构阶段。

本章第二节主要面向形成论讨论，第三节主要面向组成论讨论。

第一节　架构师的能力结构

一

架构作为一个实施对象，是有明确的实作和理论上的好坏的，并且它必将作用于一个以现实系统为对象或需求的架构目标。而架构师是以组织整体及其决策过程为背景的、实施活动中的角色之一，因而首先是以组织行为为核心的，其次才是将“架构”作为目标的优劣判别。

决策过程具有两个方向上的问题，其一是以架构目标为对象的，其二是以组织行为为对象的。

因为架构目标的特点不同，因此这两个方向并非恒等。另外，即使对于同一组织的、同一架构目标，在不同架构阶段对方向上的平衡也存在不同。

当“架构师”是一个个体而非团队时，我们可以忽略组织行为的影响。这种情况下，“架构师个体”能够全力以架构目标为对象来进行决策过程。架构意图是这一决策过程的主要出发点，而架构意图中的内在动律与整体动向为决策提供了基本的依据——一部分是事实，另一部分则是判断。

如果架构可以由“一个人”来做，那么由架构意图驱动的架构决策过程将会相当完美。而这个“架构师个体”也必因上述的原因，只需要在架构与其相关领域中有丰富的经验与技术能力即可完成这一过程。事实上，这是软件产品开发中的常态：一名架构师决定整个的系统分析、架构与设计过程，并负责在这一软件的后续产品阶段中对这些原始决策加以修正。

这时，架构师的个体能力往往决定了一个产品实施过程的推进。这一模式可以应用于大多数的软件产品开发过程中(1)，不过需要注意的是：在一些情况下，这样的架构角色也被称为（更高级别的）开发工程师。如果我们并不纠结于称谓，我们事实上会发现许多开发工程师都面临“架构决策”这一过程，因而也需要具有架构师的思维与能力。

二

我们接下来要讨论的是三个问题：其一，是否需要更加复杂的模式（如架构师团队）来推进架构；其二，复杂模式下的决策过程有何不同；其三，架构意图在复杂模式下的效果。

第一个问题的关键在于我们对“系统”的规模的设定。我们此前讨论过：作为一个“规模”的用词，系统是一个“领域集”；即使将这一领域聚焦到“数据+算法”这样的软件开发本质工作中，（在大型系统中）也被具体分成多个领域了。一旦在系统中出现跨领域和领域细分，也或者说这样的背景就是我们将“系统”作为一个规模设定的本质，那么架构也就通常是一个人无法完成的了。

因此，在这个系统的解决方案——某个具体的项目中(2)，团队中需要一个架构团队来处理架构方面的具体实作：实施与推进。但是进一步的问题是：这个架构团队应该由怎样的一些成员组成呢？对此，我认为他们——架构师应当具备的能力包括图 4-11 所示三个方面(3)。

图 4-11　架构师能力的三个方面

所谓领悟，主要包括架构思维的三个核心能力：概念抽象能力、概念表达能力和基于概念的逻辑表达能力。架构师的思维方式，决定了他在架构团队中的独特价值(4)。而这一思维方式的基点必然源自他的概念抽象能力，亦即他对“架构对象”的独特认识(5)——正是因为对其认识不同，进而才决定了对其抽象不同。

所谓领域，是架构师在目标系统中的背景知识。架构师需要相当的背景知识，才“能够”对目标系统进行恰当的概念抽象，也才“能够”准确把握系统的内在动律与整体动向。因此，

领域能力也是架构意图能够作为抽象概念与决策条件被提出的基础。

所谓领袖，是架构师在领域内和团队内的影响力。领袖能力与领导能力略有区别。后者（即领导）主要是在组织视角下对管理者（manager）这样的角色，在其职能、责权与实施能力上的说明；尤其重要的是，就组织的必要性来说，是希望限制领导角色的影响力的范围的，跨责权范围的影响力是对领导职权的一个质问。而前者，即我们这里讨论的领袖并非是一个组织角色(6)，而是指架构角色所形成的、超出组织结构的影响力(7)，其主要表达为方向、决策和对团队向心力的把握。

事实上我们所讨论的这一模型由思维、知识、行动三个方向的能力构成。总的来说，个人能力的不同取向决定了他在组织中的职业倾向，而架构师所需的是在三者中相对平衡的一种整体能力，如图 4-12 所示。

图 4-12　个人能力取向与职业倾向的对比

当从领域专家这一方向上衡量时，在领域方面的背景知识（一定程度上）反映了他可以应对的架构的规模——之所以选择从领域方面进行考察，是因为“领域”是在架构师团队中进行分工的一般标准(8)(9)，如图 4-13 所示。

图 4-13　领域背景知识与可应对的架构规模的关系

但就这三个方面总体而言，（在应对同一事件时）应当是相互支撑的，亦即是要求三个方向上的能力在整体上是平衡的。举例来说，如果架构师偏向于强化领悟能力而弱化其他，则由于在领域能力上的缺失，其架构思维趋向于理想化，偏于学术；又由于领袖能力的缺失，导致他在决策过程中丧失发言权，或有言无行，疏于实作。类似的，片面强调领域能力，则与工程师无异；片面强调领袖能力，则必将碌碌而难有所为。因而团队成员仅仅符合图 4-13 中对“领域”这一方面的要求，是不足以承担相应规模的架构师职责的。以“系统架构/平台架构”(10)为例，他应当具备与之平衡的领悟与领袖能力（如图 4-14 所示），从而避免因领域能力过强而滞于领域专家的角色。

图 4-14　架构师能力是对三方面能力的平衡性的要求

当“架构”被作为计算机系统的一个领域时，该领域也必然具有自己特定的知识，也必然具有自身的系统性需求。因此，“架构整体”作为一个系统性的目标，仍然是存在自身在“目标、规模与实现”三方面的需求(10)，仍然需要架构角色。这一角色通常被称为“首席架构师”，负责“架构整体”的决策，其能力结构仍然是我们谈到的这三个方面：在“架构”这一领域中有着丰富的知识，具有强大而独特的领悟能力，是团队中的领袖人物(12)。

三

第二个问题，复杂模式下的决策过程有什么不同呢？

首先，“架构师个体”的决策过程是简单而粗暴的。但这一过程易于成功的原因在于：架构师总是有足够的时间来推进架构，并有“零机会成本”来修正决策过程。举例来说，架构师在考虑一个决策的时候，总是可以近乎直觉地觉察到“这一架构是否可以在有效时间内完成”。但这既然是他的主观判断，那么也就可能在客观环境中出错。然而当只有“架构师个体”时，他在实施过程中提出“变更某些架构特性”的机会总是存在的。

更特别的情况下，当架构角色、设计角色与开发角色是同一个人的时候，他几乎在任何时候都可以这么做。因为架构特性是关于系统而非关于产品的特性，所以除非产品目标是系统本身（如平台化或系统改造类项目），否则架构师对架构特性的取舍与修改具有足够的发言权。从这个角度来说，这类变更（决策过程中的关键行为）在“架构师个体”而言几乎是自由的。

但是在“架构团队”中，变更的成本将会变得无法预期。极端的情况下，一旦项目开始实施，整个架构团队甚至没有机会再来修改架构。因而整个项目基于一个“并不良好的架构”来开发实施的情况，是必然存在的。

种种因素导致对于“架构整体”的决策并不是在讨论学术上正确与否，而将是讨论是否能够基于现有团队实施推进。这涉及团队管理中的几个问题：

（1）对架构师团队与技术团队的评估；

（2）适时地中止讨论并形成架构决议；

（3）对实施过程有效跟踪并适时发起调整过程。

另外，参考格拉斯问题解决模型(13)，整个架构推进过程还涉及两个时间方面的决策：

（1）何时能确定架构解决了系统的核心问题并可以进入实施推进环节；

（2）一旦实施中发生变更，确定该变更应当在何时予以满足。

上述五点（但不仅限于这五点）提出了在“架构整体(14)”上所需要的决策过程。其基本模型如图 4-15 所示。

图 4-15　在“架构整体”上所需要的决策过程

图 4-16 说明将该模型应用于一个开发实施场景中的具体架构决策过程。

图 4-16　多个实施阶段下的架构决策过程

这一复杂的决策过程是由多个架构师角色参与的，但其决策者必是其中“以架构整体为目标”的架构师。这些决策确定了架构的整体走向与实施规划，以影响架构的整体性为基本目的。但既然参与者众，而决策者寡，则必然涉及决策者与众不同的决策能力的问题。

四

第三个问题则是关于架构意图在复杂模式下的效果的。这涉及我们上面讨论的一个核心设问：架构整体需要决策的本质原因是什么呢？

我们所讨论的系统的规模已经扩展到多个领域，因此需要由架构师团队来处理它的架构需求。进而地，多个领域之间的——系统本身的——问题作为一个独立领域仍然有自身的架构需求，因此我们提出了系统架构师或平台架构师等规模来应对之，并（根据其领袖能力、可能性地）赋予其一定的组织责权，例如“首席架构师”或“主架构师”。

这一架构角色面对的并非上述系统各个独立领域内部的问题，而是“架构整体”的问题。将其本身作为系统，综合一下我们此前的讨论。

□　其一，它是领域集。从领域集的关系来看，它可能涉及的基础构件包括领域/子系统、通信与验证，以及它们的问题与解决方案，例如分布、依赖、消息等。

□　其二，从其定义来看，它的抽象必须能容纳上述构件并提供可以讨论的事实基础。

□　其三，从系统性的限制上来看，上述事实基础必将涉及全局性的边界约定，以及对非可控因素（包括但不限于风险）的识别与处理。

□　其四，从架构的本质上来看，其范围与联接件的问题，实质上是各领域边界的全集与交集（以及交集间）的关系。

可见，这些内容提出了“系统需要事实基础”，这与架构意图所讨论的问题是一致的。

但这并不能表明“架构本身的架构意图”与领域性的、子系统的架构意图有着一样的源起与价值——我们在这里提到了“价值”，亦即是对“架构本身的架构意图”的效果问题的讨论。

架构有两个效果方面的考量，即它对时间需求与空间需求的响应与收益。这样的考量依据来源于以下三点。

□　其一，若架构不谈时间需求与空间需求，而只谈目标需求，那么“架构整体”就必将等效于“各种架构的全集”。然而，若这个全集的元素之间没有关系，也就无法构成整体，进而“全集”这一观念构成了对架构整体性的破坏。

□　其二，如前所论，架构是可以通过解决问题来实现需求的，而非单纯对需求的响应。若

架构本身只谈上述全集的“目标需求”，那么也就无法触及其背后的“问题”；而时间需求与空间需求背后的问题是清晰的，即系统的规模与复杂性。

□　其三，架构本身的价值在于：在保持方向的同时控制成本。而架构在时间需求与空间需求上的考量，构成了“架构全集”到“架构整体”的价值提升。它使得架构可以通过在时间与空间上的分解——一般表达为架构阶段（以及对应的实施阶段）的迭代——来解决架构规模问题与复杂性问题，进而达到成本控制。

综上，团队模式下的决策与个体决策有很大的不同(15)。团队决策考虑的对象有两点，其一是对架构整体的把握，其二是对团队整体的把握。对前者的思考，仍然可以归于架构意图，是由领悟能力驱动的；而后者则可以视为对架构意图的效果的保障，是由领袖能力所驱动的。

第二节　系统架构与决策

一

“架构师团队面临的架构整体”是什么？我将它称为系统架构。这缘于这一“架构整体”是对系统整体的抽象。然而需要强调的是，对于系统架构这一指称，其中的“系统”是一个狭义的、表达开发规模的用词，同时它也表达上述规模下的开发活动整体。

针对系统架构的架构意图，我们仍然可以提出如下设问：

（1）其一般过程是什么？

（2）其可能的演化方向是什么？

（3）该系统对于客户战略作何种响应？

（4）什么是系统的本质问题？

（5）能不能不做？

在继续讨论之前我必须再次强调：接下来的讨论并不基于该系统架构之局部（例如子系统/应用的架构），而是面向其本身——系统架构。

二

任何系统架构必存在其外部实现与内部实现的过程。所谓外部实现，即是指架构师团队用以形成与演化架构的过程，以团队决策模型为例，即是本章第一节中基于架构师团队所讨论的

决策过程(16)。所谓内部实现，即是架构以及其部件的内部关系得以构建与维护的过程，以架构形成模型为例，一个可能的过程如图 4-17 所示。

图 4-17　一个可参考的架构形成模型 M0

这张图已经表达了一般过程中的限制条件与流转关系，但仍然需要强调两点：其一，在“实现架构”与“开发架构”中，分别只列举出了其中最重要的两个组成部分，这并非其全部；其二，在“实现架构”中只列出了运行架构与集成架构的原因是，它们对部署与开发的约束作用最为明显。

从上述一般过程的各个关键环节来分析，一个架构的有效性、正确性应当表达为：

□　如何确保宏观规划层对需求映射层的约束，以及确保功能架构对开发架构的约束；

□　如何确保在将能力架构映射为实现架构时不丢失功能设计；

□　如何确保开发实现的结果能够被应用于预设的交付环境。

以此为参照，我们回顾此前图 4-7 所示的“通用办公系统架构　v0.0.0.2”，为方便阅读，重复于图 4-18 中。

图 4-18　通用办公系统架构 v0.0.0.2

确切地说：它充其量只能算功能架构(17)，离系统架构还很远。但是有趣的是，在一般性的“办公系统的规模”下，从“架构　v0.0.0.2”开始就已经可以进行有效的软件开发活动了。

从上述过程来分析“架构 v0.0.0.2”的形成过程就会发现，事实上得出“架构 v0.0.0.2”模型时，架构师——比如我——就已经隐含地：

□　完成了对业务及其需求的分析，如架构 v0.0.0…x 到 v0.0.0.1 的整个过程；

□　预设了这个办公系统的规模，例如在此前的分析中提及的“不需要跨国管理”等限定条件。

也就是说，“从架构 v0.0.0…x 到 v0.0.0.1 的整个过程”正是上述一般过程的一个应用示例。这也就是它仍然可以用于（通过后续的、持续的细化来）推动开发活动的真实原因。

“架构 v0.0.0.2”太过于简陋，而且也未能在“系统架构”的背景下来表达架构意图，所以无法表现上述一般过程。撷其片段，可以就“这是一个什么样的系统架构”暂作一小结，其架构意图可以表述为（通过上述方法与过程，将最终构建）：

以功能性为核心的管理系统（的架构）。

三

现在，让我们再前进一步(18)，将这一架构意图表达为一个概要的系统架构的模型(19)，如图 4-19 所示。

图 4-19　一个概要的系统架构的模型

我们需要在后续的架构活动中补全它对于开发与部署的约束性，由此得出整个“架构团队”的完整工作集(20)。其中，功能层是不需要过深讨论的，因为它应当可以通过“通用办公系统架构 v0.0.0.2”逐渐细化而来，表现为一种功能架构（functional architecture）。

服务层可以用一种静态的运行架构（static view of process architecture）(21)来表达，例如将不同的功能包装并发布成服务，如图 4-20 所示。在图 4-20 中，角色定义服务是对功能层中的“角色集”的封装，而业务登记服务是对“功能集”的封装。“定义”与“登记”在一定程度上暗示(22)了两种封装的形式不同，前者可能是一个角色管理子系统，后者可能是一个调度框架中的注册模块。

图 4-20　服务层的模型

框架层是一种更高层级上的架构决策，它讨论驱动上述这些服务与功能的整体方式。亦即，框架决定了功能层运行在何种环境中，也决定了服务层中的各服务之间的结构关系，甚至也决定了整个功能、服务层的入口以及其后整个交互。当然，框架层的另一个有趣之处还在于它决定了用户——产品的使用者与系统打交道的方式，即接口。

框架是一种动态的运行架构（dynamic view of process architecture）。运行架构被框架层和服务器分为了动态与静态两个部分，这取决于你以何种视角来观察这些部件。例如，因为出现了“业务集”这一概念，所以所有业务可以视为静态的被组织单元，而其组织方式为“（业务）登记”；更进一步，框架为这些静态的业务而准备的机制可能就是“调度”或者“事务驱动”。这样的一个框架层，可能会表现为图 4-21 所示的架构。

图 4-21　框架层的模型

注意框架的“可运行特性”，这意味着框架必须能描述数据和（阶段性地）持有数据的逻辑之间的关系。上述架构中使用了箭头，用以指示某些信息流的流向（call）与转换（transation）。

我们必须保证上述框架层在数据与逻辑关系上是完整的、足具的，否则框架本身便无法实现“动态的运行架构”这一目标。

产品层依赖集成架构（integration architecture）来表达——系统集成活动的输出通常是一个“产品”（product），并且其输入也依赖（其他的）产品。例如，系统运行依赖运行环境、数据库，以及具体由开发团队开发的代码包。我们的所谓集成活动，并非简单地将代码包 build 起来，而是指将代码包 build 成一个产品，并确保它与运行环境、数据库这些外部产品能配合起来工作。这个最终配合无间的整体，才是我们的系统，也才是对用户而言有意义的产品。

对于上述系统，其最终的集成架构可以用图 4-22 所示的产品层的模型表达。

图 4-22　产品层的模型（面向产品的集成架构）

这个架构表达了维持框架层正确运行所需的环境、工具与测试脚本等部件。这些内容/部件中的一部分是开发活动所需产出的产品，一部分是第三方产品，一般来说，后者是首选。大多数情况下，集成架构表达的是需要集成的产品内容及其之间的关系。此外，因为集成架构的内容是产品而非程序的调用接口，所以它们之间的关系将主要是组合、依赖与层次等，而非调用或数据返回。

四

通过产品、框架、服务与功能这样的层次，系统架构整体地表达了它对实现域和交付域（以及阶段）可能构成的影响。再来看一下图 4-17。

□　功能架构：将功能分割为基本独立的功能块，基本映射了用户的原始需求，并约束了开发架构中的功能模块。

□　运行架构（静态部分）：将这些功能包装并发布成服务，用以约束开发架构中的包的组织与接口的设计。

□　运行架构（动态部分）：选择或实现可运行框架来驱动服务与功能，基本约束了开发架构中可选的第三方应用服务器，以及应当自主开发的、系统中的关键联接件，如事务服务框架等。

□　集成架构：以产品来封装和交付可运行框架，基本约束了部署架构可用的部件，以及部件之间的组合、依赖等关系。

综上所述，系统架构可以通过上述的一般过程来完成决策，最终将在不同的阶段产出相应架构（一般称之为某种架构视图）来影响其他阶段的工作。在产生这些架构视图的过程中，隐含了大量的架构决策细节。例如：

□　功能架构中，账户子系统与部门子系统可以依赖 Windows 中的目录服务（直接使用或二次开发）；角色定义服务，意味着它需要依赖某种对定义（definition/specification）的解析服务，也就是说，该服务应当基于某种策略服务器来实现。

□　运行架构的静态与动态部分，事实上是参考了 JavaBeans 模型。因此如果使用 J2EE 服务器，则应用服务器、Beans 容器及其运行框架，会话、事务等基础服务，以及在架构的各个层次上都有较为成型的解决方案。

□　集成架构主要表达产品的规划以及产品之间的组织关系，这与系统的“领域集”特性有关，也与系统规划（包括上述的框架选型）带来的可组织性有关。

由此可见，“架构形成模型 M0”的提出，提供了“通过什么来影响什么”的一般过程，进而对种种架构内容与阶段提供了参考。我们可以顺着这一过程来梳理架构活动，进而形成或明确我们在系统架构上的架构意图。如上一小节的示例中，已经渐行渐显地得到：

一个以企业管理应用为目的、以业务功能为核心的三层架构。

需要补充的是：我们在现实中的架构活动，通常是先“凭经验选择”了某种架构模式（如三层架构或多层架构），然后在此基础上进行框架和第三方产品的选型，最后再按照这些基础环境的约束来做自己的架构工作。——这一过程通常如此，但它只是一种可重复的实现方法，而无法解决“如何形成架构意图”这一问题(23)。

最后，对“架构形式模型 M0”作一些简化，得到图 4-23 所示的系统架构的一般过程。

图 4-23　对“架构形成模型 M0”的简化：系统架构的一般过程

在上面的例子中，我们事实上是以能力架构为出发点来讨论实现，而缺少对体系架构的思考的。因此，我们提出了“如何定义实现架构，以使它满足系统的能力需求”这个设问，并基于“架构是在不同阶段形成的（即架构形成论）”这一假设，通过“一般过程”来探求系统架构的实施与决策(24)。

五

基于一般过程的设问，无助于讨论系统架构的整体实施走向，即“将向何处去”的问题。关于向何处去的问题，我认为若架构本身是应对规模问题，是面向领域集的，那么系统架构的演化方向必然只有两种：要么更大，要么更小。我常常将这两个方向称为“大到看不见”与“小到看不见”。

一个架构总是对它的构成部件在边界与联接件两个方面的设定。所谓设定，即是明确边界的范围，或明确联接的方法。然而，架构的主体——系统本身，却是动态地基于现实系统而演进的。如果我们有一个极端明确的边界约束，例如“由 A、B、C 三个部分构成”，那么当系统需要加一个新的领域时，架构就失效了；如果有一个极端明确的联接方法，例如“必须让

A 成为 B、C 的中间环节”，则系统中将不可能容纳未知的 D，因为 A 不能预知 D 与 B、C 的关系。

我们似乎是在讨论“无穷边界与关系”的系统，但系统架构对应用与子系统（以及其相应业务）的“可容纳性”决定了这必然是系统架构的方向。因此，就“系统架构”这个领域出现的本质来看，它就应当具有两点特性：

□ 它能反映系统长期演化中的不变性，以在演化过程中持续用于对系统的讨论；

□ 它不能是系统阶段实现的负担。

显然，系统架构的作用与其方向上构成了一对矛盾。但是在我们的实践中，这个矛盾是有解的，亦即所谓平台（platform）与框架（framework）。

这两个解，也是对系统架构中的“体系架构”的两个抽象(25)。首先，架构的支撑性应当以数据为核心，也就是说，平台通常是围绕数据的位置、功用、生存周期或其分布特性来规划的，例如常提到的三层结构在本质上就有平台化的倾向，因为它明确了交互数据、应用数据与系统数据在三个层次上的位置，以及相互间的产生、转化过程。其次，架构的调度性应当以逻辑为核心，也就是说，框架应当追寻架构对象——系统——的一般过程，并将它实现为架构的核心调度逻辑，例如 COM 框架，其核心就是组件的 register-request 这个过程。

若系统架构以平台为方向，则应当力求“大到看不见”；若系统架构以框架为方向，则应当力求“小到看不见”。所谓“看不见”，就是指该架构的存在不应当对系统的其他部件（例如对应于不同的领域的子系统）的实现构成影响。我们做架构的目的并不是要做出一个东西来阻碍我们的开发实施工作。我们的现实目标是“做一个系统”，而“系统的架构”是用来讨论系统的一个工具；如果这个工具最终影响了系统的形成，影响了系统本身，那这个工具便失去了原初的价值。

无论是做平台，还是做框架，最终的目标都是让系统基于它或使用它，而又无视它。

六

我们此前提出的架构意图：

一个以企业管理应用为目的、以业务功能为核心的三层架构。

真的具有平台特性吗？真的是以平台为方向的一种架构意图吗？

答案是：未必。因为我们此前通过“架构形成模型 M0”提出的这一意图，是基于系统构成的，而上一小节讨论的平台/框架则是面向系统的方向来讨论的。尽管两个讨论中都用“三层架构(26)”为例子，但实质上是不同的。

决定“系统架构的架构意图”的另一关键因素是对客户战略的把握。仍以“办公系统”为例，这里将至少涉及两类客户(27)，其一是系统的实际用户方，如某个公司的某个职员；其二是

以“办公系统”为可销售产品的、开发者所在的系统开发方。为了后续的讨论，我们简化地做出点假设：其一，主要的影响来自于用户方与开发方的战略；其二，上述战略要求办公系统能够长期而持续地添加或变更需求。

在战略决策过程中，“目标系统拿来做什么？”会是一个关键问题。它讨论了开发以及持续开发的必要性，后者更是进一步地决定了“以系统（这一规模下的）方法来架构它”的必要性。图 4-19 所示（重复于图 4-24 中）的架构模型是对上述问题的一个不错的回应，它意味着这个系统是满足“（用户方）功能渐增”这一需求的。当功能增加时，只要这些功能可以被抽象为服务并置于框架层中，那么它必然可以作为产品的一部分发布或投放。J2EE 这一企业应用的解决方案本身作为一个系统实现了框架层与服务层的统一，并且在框架层与服务层提供了系统分布的解决方案。这样就一举解决了三类需求：

图 4-24　一个概要的系统架构的模型

□　用户方在功能渐增上的需求；

□　开发方在产品封装与发布上的需求；

□　系统在（通过分布与部署来解决的）规模性上的需求。

但是这三点表现出来的是，“框架层+服务层”对系统运行逻辑的约束，而非对系统在数据性质上的规划。因此这一方案以及由此形成的具体应用解决都是“框架”这个方向下的。

那么“用户方、开发方与系统”对于平台的需求为何呢？

平台是用于整合资源的，这是由平台本身“面向数据”这一特性而决定的。如果用户方或开发方具有整合资源的需求，无论这一资源是上下游的供应链（行为、活动或运营模式等），或是系统供求关系中的依赖物资（用户、时间或实际生产资料等），又或者是在多种资源之间通过某种方式转化（授权、交易或数据挖掘等），那么它都需要一个平台来描述这一资源的核心生存周期，并基于这一核心生存周期中的一个或多个阶段来构划平台。

平台的核心在于支撑，这意味着平台（或平台中的层次）对数据的持有是独占的——在同一平台中对数据的理解是一致的。如果不具有这种特性，那么应当增加一层数据抽象，并在该层次上再构建新的平台。

仍以办公系统为例，从用户方来看，一种提出“平台需求”的理由是(28)：办公过程中的审核行为可能会涉及对多种外部活动（如工作流中的环节）的确认。那么这种情况下，审核对象应该可以理解为跨子系统的统一资源，因此应当通过对审核对象的生存周期（产生于何子系统，在何子系统中使用等细节）进行规划，并提出“平台化审核”的需求。这样一来，办公系统的架构意图就倾向于平台方向了，我们可能看到在将来的实施中，在办公系统中去整合工作流系统或具体业务系统，也可以整合决策系统等。因为从这些系统的特点来看：它们基于管理层次，通过审核来完成行为注册、提交、授权与验证等功能。

从开发方来看，也可能存在提出“平台需求”的理由。例如，开发方试图将“为企业定制办公系统”过渡到“开发通用办公系统，并提供企业定制服务”。如果有了这种需求，那么办公系统本身将需要被平台化，以整合其上的资源——各个“办公模块（子系统）”。

J2EE 这样的架构模型并不能很好地应对平台需求，因为它在核心上只是将服务理解为资源，这是“计算机系统”的视角而非业务视角。但是反过来看，这也意味着这样的架构模型在技术实现方面是可以平台化的，只是架构师需要从业务视角上对平台进行架构意图的展现、明确与推动而已。

三层或更多层，并不是平台化。层次是平台化的一种表现方法，而非平台——作为架构意图的本身，也并不是平台在应对战略问题中的核心关注点。

七

什么是系统的本质问题？这是我们要明确讨论的最后一个关键设问。就题设来说，“系统”的本质是领域集。这是一开始就讨论过并强加在我们这里的讨论之上的。但其本质上的问题是什么，却是一个还没有被讨论的话题。

若以现实系统为各个领域的目标，那么系统只需实现目标需求即可，亦即本质问题将是“能或不能实现”的问题。但是这将会得到一个“死的”系统：从系统被完成的第一时间开始它就不再增删任何东西；也没有任何对外的接口，因为它不面向新的领域。这样的系统并非不存在，事实上它大量存在着，并且是“高可靠系统”的主要开发方式。这样的系统的一个主要特点是它在架构上的确定性，与之对应而又匹配的，则是其领域集中的运作模式也相对确定。

但我们是在讨论复合领域集的问题。尤其其关键核心在于：由领域集构成的业务模式（犹如产业链等）是可能变化的，甚至是变化频繁的。一如本章最开始所讨论的，我们将“面向时间需求与空间需求”来进行架构，以应对这种持续变化。更进一步地，“变化”本身也只是需求，背后真正的问题——架构的本质问题，其实是对系统性的维护。

总的来说，可以将此前提及的种种思想归于在这一问题下的求解。例如，以架构形成的一般过程为参考，或架构以平台或框架为方向，或架构对战略的响应，等等，这些无一不是在讨论这样一个问题：架构需要提供何种程度的持续性，以使得它能够应对系统在构建和应用过程中的变化——并在这些变化之下保持“系统整体的一致”？

抛开上述这些具体方法，直面问题本身，那么架构规划也可能是另外的一种求解思路。仍以办公系统为例，从中长期来看，开发方推广该产品会存在一些明显的阶段。例如：

□ 在早期可能应用方单一，所以系统功能明确，针对性强而定制性弱；

□ 在一段时间之后，系统的应用方就多了，但缘于业务开展的情况，所以用户大多数都在类似行业当中，因此功能的定制性要求虽然强了，但其领域性仍比较单一，并且大体的使用流程和现场问题都类似；

□　再后，开发方可能要求该系统平台化——这基于两点要求，其一是业务推广开始面向通用领域，其二是功能的大量增加导致系统过于复杂；

□　最后，整个系统还可能走向跨平台的方向，例如面向不同的终端（如手机），以及加入不同的设施（如办公室签到设备），或者跨地域的办公等。

我们可以对这些阶段作出规划。首先，不同阶段的系统重点是不一样的，如表 4-3 所示。

表 4-3　架构规划在不同阶段中的重点

当我们将当前系统的目标与上述规划对应时，就很容易锁定我们“应有的”架构意图，并且能够通过阶段规划，来促使当前的架构意图契合业务方向对架构的要求。例如，就办公系统而言，我们可能设定为：

架构一期，快速实现的市场探针性产品。

基于这样的设定以及对其后的持续性的考虑，我们就有了进一步的架构决策。例如：

□　宜通过使用现有的成熟系统来搭建运行环境，加强现场团队的实施能力来解决客户的现实问题；强化现场团队的反应能力、反馈能力，对发现问题的敏锐度，以便更加准确地收集与响应客户需求。

□　宜通过敏捷开发团队的模式来实现产品功能，可以考虑让客户直接参与测试过程（例如试用或小范围测试），可以直接向客户提供 alpha 版本。可以考虑客户配合的现场调试环境。

□　简化产品化特性，例如说明书或安装包。

因此，在这个阶段的架构上，对集成性、重用性、移植性等的考虑就会非常少，类似集成架构、功能架构等的实施也可以很概略。

但是从架构的体系性上来考虑，即使在最初阶段的架构中，也不能缺少对后续架构阶段的设定。这至少包括两个方面：其一，后续架构阶段的启动条件；其二，后续架构过程“在当前架构中的”入手点。表 4-4 给出了一个示例。

表 4-4　架构意图在不同阶段中的变化与入手点

架构意图在各个阶段的不同变化，意味着我们可以用“有规划的变化”来讨论整体的架构意图。我们需要做的最后的一步工作是：将几个关键入手点连续起来，于是整个系统架构的脉络也就赫然可见了。

第三节　架构的表达与逻辑

一

在与朋友的饭局中，我常常被问道：你能吃辣椒吗？我的回答一般是：我是四川人。我从来没有因为这样的问答而在饭局中遇到尴尬——朋友们都能欣然选择一些口味偏辣的食物，毫不犹豫。

好吧，我承认我很喜欢辛辣食物。

但是我很多次反思过这个问题，即：“我是四川人”这个答案到底表明我是能吃辣椒呢，还是表明我不能吃？就其逻辑来说，这个答案并不能表达我的饮食偏好；然而就彼时彼事的实效而言，却没有人误解。

可见所谓“暗示”，在日常生活中常常是有效的。但是，这首先应当基于双方有类似的阅历背景，例如：

□　“能吃辣椒吗”这个提问在饭局中通常并不是询问你能否直接食用单个辣椒，而是询问你对辛辣食物的接受程度。

□　“能吃”对程度的表达是很模糊的。能吃多少，以什么样的形式吃，这些信息都不能在设问中得到。通常，如果当时是在川菜馆，那么回答“能”的人多少心里要犹豫一下；而如果吃的是杭帮菜，作这样回答的人就多了些底气。

□　“我是四川人”这个回答是说给了解四川人的普遍饮食习惯的人听的。如果面前有几位外国朋友，那么他们通常会对上述的问答茫然无解。

□　问答双方其实都了解这样的一些事实，即“我是四川人”这个回答是在暗示“我与大多数四川人一样”，而非陈述我的籍贯；并且，结合当前的环境，所谓“与大多数四川人一样”的性质是针对辛辣食物而言的；最后，就这一性质而言，普遍性的情况，亦即是事实是：偏好辛辣食物。

如果上述是一个推理过程，那么整个过程包含了许多背景知识和事实推定。其中最重要的一项假设是：我认为提问者与我有相同的知识背景，以及能够做出相同的事实推定。另外，也正如这个例子所提及的，我们在某些情况下其实可以接受一些“程度模糊的”信息。在上面的问答中，最终辣一点或淡一点，都无妨整个饭局的质量；问答者之间，本身也都没有对“何种程度的辣”有一个同一的标准。

与此类似的，我们的架构过程中也有一些信息很模糊。例如，我们在讨论“办公系统是一个什么样的系统”时说：

……我们需要更深层次地设定“被规则化的”的这个系统本身。总结我对这一设定的考虑，它将会是：

□ 与现实系统看起来类似的、

□ 具有同等的组织容量的、

□ 基本符合现实系统的运作逻辑的

一个软件系统。

注意，在这里用到的类似、同等与基本（符合），都是很模糊的程度用词。

所以，总的来说，我承认架构过程中存在大量的背景知识和推定事实，并且信息表现得可能会模糊而含混。这是因为架构过程本身就是对目标系统中一些不太明晰的概念（边界、联接关系等）渐次清晰的过程，所以架构过程中的模糊信息是必然存在的，否则根本不必去架构它。

但是这并不妨碍我们要求对架构的表达必须清晰准确(29)。因为实施过程必将依赖架构的结果，亦即是架构的最终表达与决策。架构表达是在架构过程的阶段性成果的基础上所进行的、尽可能准确而清晰的叙述。如果它不能尽量准确地反映架构过程的阶段性成果，那么它也就不能作为下一个阶段（无论是实施还是新的架构迭代）的有效依据。换言之，如果对架构结果的表达是模糊的，则该结果是无意义的。

那么，让我们从一个最简单的表达开始。请问：在白纸上画下一条线，意味着什么？

二

无论如何，在架构图中出现了一条线，通常都意味着它将一个整体划分成了两个部分。习惯性地，我们用纵向的线来表明领域的划分，而用横向的线来表明层次的划分，如图 4-25 所示。

图 4-25　整体划分成两部分

由于不同部分自身“方框”的存在，因此这些横纵的线条也可能不被明确地划出。但就其含义上来说，它们都是存在的。例如，图 4-26 所示的“Spring 架构图”中隐含的一些领域与层次信息。

图 4-26　Spring 架构图

无论是用线来隔开两个部分，还是用两个方框来表达这两个部分（进而由方框的边界来区隔它们），当我们表达出两个或多个部分时，每一个部分都需要一个明确的概念来指示。这涉及两个信息：其一，各个部分之间的分类依据；其二，各个部分所需的抽象指称。仍以上述“Spring 架构图”为例，图 4-27 中的“数据对象域”与“一般对象域”是二者的分类依据(30)，而 ORM 与 JEE 是二者各自的指称。由于 ORM/JEE 这样的指称一定程度上也暗示了分类依据，因此（在交流双方或团队具有相同的知识背景的情况下），也可以省略上述的线与分类依据标识。

图 4-27　Spring 架构图：ORM 与 JEE 之间的领域与分类依据

在架构图中，横向的线也会有类似的表示法。例如，图 4-28 中 AOP 与 Core 是各自的指称。“核心”（Core）这一指称在一定程度上具有“支撑”（外围）的语义，并且其注释中的 container 是容器框架的含义，这两个信息表达了它在对于其上的 AOP 等内容的支撑与包含关系。这些“暗含性的指称”使得在框架图上去除掉一些线（以及对其分类依据的标注）之后，仍然有较明确的含义(31)。

图 4-28　Spring 架构图：AOP 与 Core 之间的层次与分类依据

如上，我们得到了一个整体的各个部分：相互间可以用线条来区隔，其自身可以用方框来表示；并且，每一个部分都“应当（且必须）”有一个明确的概念，并有文字来指称它。

一个部分所包含的子域应当位于它的方框（当前域）之内。这些子域的概念应该派生自当前域，是当前域的概念的细分、子集、关联或延伸。有时为了图示的简洁，也可以将子域的指称直接置于当前域（的方框）之中。例如，图 4-29 中给出的两种图示具有相同的含义。左图中直接包含了一些子域的指称（列表），注意它只强调当前域对子域的包含关系，并不强调这些子域之间有何种分类依据或存在限制关系；而右图一定程度上会导致对子域之间是否存有层次性的误解。

图 4-29　Spring 架构图：JEE 的两种图示

因此，如果不存在层次关系，应尽量避免右图的表达方法。

三

如果我们

□　用一个方框来表示领域，并且

□　把一个方框分成两个（或多个）以表明领域之间没有关系或仅有殊少关系，

那么当我们试图在一个平面上来表达这些方框时，（依我仅有的知识来看，）大概只有三种方法，并进而得到图 4-30 所示的三种结构。

图 4-30　在平面上表达领域的三种方法

如前所述，当我们试图在两个部分之间制造一个界面（如画一条线）时，我们是需要讨论这两个部分之间的分类依据的。就分类这一行为本身而言，我们可以有无数种依据以及无数种方法，但就我们这里需要讨论的问题——如何降低或至少不增加系统整体的复杂性来说，一种可选的分类依据是：如何隔离变化。

系统的复杂性有很大一部分是由其可变性导致的(32)。但既有其可变处，也必有其不变处。以上述三种结构的表达方式来看：

□　如果一个系统的公共部分是不变的，那么它适合用层次来表示；

□　如果一个系统的总量是不变的，那么它适合用并列结构来表示；

□　如果一个系统的核心是不变的，那么它适合用嵌套结构来表示。

以层次结构来论，如果我们能从系统中捕捉到那些不变的公共部分，我们就可以将它表达在底层，反之将“目前看起来可变”的部分表达在上层。如此，在一系列的架构活动结束之后，我们总是能保证系统的基底部分是无需变化的，亦即它是稳定的；相对于系统整体来说，它带来的复杂度应是衡为“1”的(33)；它决定了系统整体的性状是不变的(34)。

就“系统架构”的整体表达来看，层次结构适宜构建平台（platform）的过程，其基础领域倾向于不变；并列结构适宜构建库（library）的过程(35)，其领域总量倾向于不变；嵌套结构适宜构建框架（framework）的过程(36)，其核心领域（或核心过程）是倾向于不变的。

四

多个方框放在一起的时候，它们（所表达的领域）之间是没有关系或仅有殊少关系的。其中，当使用并列结构时，它通常表明系统总量是不变——系统的复杂性不因为拆分而增加。这事实上也约束了并列结构之间是不应有相互关系的。因为并列结构之间若存在关系，则“处理这些关系”将带来系统本身的复杂性的增加，而这与我们使用并列结构的本意是矛盾的。

当并列结构之间不存在关系并且它所表明的系统总量不变时，并列（的所有域）是可以被视为一个整体的。换言之，拆分或不拆分，只是对规模的分解而不会导致关系或相关处理的增

减。就系统整体来说，其规模将因 A..Z 的个数的增加而线性增长，但其复杂性仍然是衡为 1 的。因此，图 4-31 所示的两个图例是等义的。

图 4-31　并列结构的表现形式

嵌套结构所谓的“核心”，是指所有除核心之外的其他领域必然与该核心发生关系，亦即它必然可以表达为图 4-32 所示形式的结构。

图 4-32　嵌套结构的表现形式

但 B..Z 之间是不是存在关系，会是一个很关键的问题——如果 B..Z 之间仍然存在关系，则图 4-32 所示的图将会类似网状，这会带来系统复杂性的剧增。所幸，在这个模型里，我们可以认为，如果 B⋯Z 的任意组合之间存在关系，则它应当视为 A 的一部分，亦即通过扩大 A 的规模来减少 A⋯Z 之间的整体复杂性。如此，以 B_C 之间存在关系为例，它的模型仍然可以表示为图 4-33 所示的嵌套结构。

图 4-33　以 B_C 之间存在关系为例改写架构的表达

这样一来，A'自身（缘于此前的设定，A'即是嵌套结构所谓的“核心”）事实上应当具有 A..C 的系统总量以及它们之间的、确定的关系带来的复杂性，但该复杂性因为关系是确定的所以是确定的，而 D⋯Z 的复杂性是确定的。因此总的来看，二者的复杂性仍然是确定值 1。

因为 D⋯Z 之间是没有关系的，所以它们也可以被视为一个并列结构 D_Z。当我们把 A'与 D_Z 看成整体结构 A'_Z 时，其复杂性应该由上述确定值 1 与一个关系 m 构成，可计为 1+m。该结构如图 4-34 所示。

图 4-34　嵌套结构与并列结构

综合上述有关嵌套结构（A'_Z）与并列结构（A_Z）的讨论：既然它们对应系统的复杂性都是确定值，那么它们都应当可以作为层次结构的一个可以“视其为具有不变性”的独立部分加以讨论。以图 4-35 为例，也就是说：

图 4-35　其他结构在层次结构中的含义

□　其一，由于嵌套结构可以理解为分成“核心与非核心两层”的层次结构(37)，因此总的来说，非层次结构（嵌套和并列）的使用并不会带来系统整体复杂性的增加；

□　其二，对于层次结构的任意两层，其层次（自身的）复杂性为 1，因此其整体复杂性是由层间（关系的）复杂性决定的，可计为 1+m；

□　其三，对于层次结构整体，它将包括不变与可变两个部分，由于其不变部分的复杂度是 1，因此其整体复杂性必是由可变部分导致的。

最后，就系统架构整体来说，我们必须关注三点：其一，应通过层次系统来隔离可变性，并尽量增大其中不变的部分；其二，可变部分影响系统的形态，但不影响系统的性状（亦即是指系统的边界与联接件）；其三，如何理解“不变部分的关系”决定了系统的性状，也决定了“不变部分的复杂度是 1”的单位大小。

五

我们还没有讨论过关系的复杂性 m。这一论题的重要性在于：如果它是确定的，则我们上述讨论的“除层次的可变部分之外的”其他所有部分——无论它是何种结构形式，都必然确定；否则“（层次结构的）不变部分的复杂度是 1”将成为伪命题，而层次结构也必然无法降低系统的复杂性。所幸，这些可被讨论的关系是有限的。这在

□　第八章　系统的基础部件，第四节系统（四）

中就曾经提出过：任意领域的系统对“当前系统”的需要只有两个，亦即寻求计算资源，或寻求数据资源。回顾此前的讨论，若 A 需要 B 的数据资源，我们称为数据依赖；若 A 需要 B 的计算资源，我们称为逻辑依赖；若将 A 和 B 都视为逻辑——可计算资源，并讨论二者之间的关系，那么我们就会看到（逻辑或数据的）时序依赖。

这些“有限的关系”的复杂性 m 如何呢？

首先，并列结构之间是不应存在关系的——这与我们此前对于并列结构的表达原则有关。如果它们所表达的领域之间的确存在依赖（如图 4-36a 所示），那么事实上可以将被依赖域下沉为一个独立层次（图 4-36b），并使其他域基于该层以使整体表达为层间（向下）依赖关系（图 4-36c）。

图 4-36　并列结构之间的依赖

层次结构中会存在向下的关系，这可能来自于两个方面：其一是上述因领域到层次的转置而导致的，其二则是层次结构的内在属性。后者缘于层次结构形成的时序性（参考图 4-35）：其上层的“层次（可变部分）”总是晚于“层次（不变部分）”而形成的，因为它们总是“目前看起来可变（的那些部分）”的一个集合。那么，就其形成逻辑而言，若未知部分不与已知部分发生——任何可能发生的——关系，则未知部分必然可以不属于系统整体，也必不对

系统整体确定性构成影响(38)。

因此对于未知部分，若它确是层次结构整体的某个部分，则必然有向下的关系。然而反之，在层次整体的“可变部分与不变部分”之间，向上的依赖是不应当存在的。因为若存在可变对不变的依赖，那么它事实上就等义于已知对未知的依赖——这与“已知”这一概念正好相悖，也必是不确定的(39)。

在“层次（不变部分）”内的各个层次间，（同样是基于时序性，）向下的依赖关系也是必然的。但是反之，向上的依赖关系则不一定。因为无论如何，既然这些层都是确定的、不变的，那么它们之间的关系——若存有相互的依赖——也必是确定的、不变的。所以就其时序性来说，层次的内在性质是容许向上依赖的。

但是，向上依赖并不会因为层次的时序性而产生——以如上的讨论来看，层次的形成时序与确定性需求构成了一个（严谨而完整的）逻辑，这导致层次间只会存在向下的依赖。我们之所以必须讨论向上依赖，是因为它可能会来自于一种相互依赖关系(40)：我们“同时”识别到两个领域，这两个领域确实相互依赖。这种情况下，我们如果试图将领域转置为层次，那么层次间也就必然有两个方向上的依赖关系了，如图 4-37 所示。

图 4-37　将领域转置为层次带来的依赖关系

我们先讨论两种时序依赖关系之一(41)，即 A 和 B 间存在相互的数据时序依赖关系。若 A 和 B 是各自依赖了不同的数据而导致这种关系，则可以将 A 中的数据抽离至 B(42)，如图 4-38 所示；或将 A 和 B 的数据都抽离出来并因这些数据已知而置于底层 Z，则可以避免上述依赖(43)。若 A 和 B 依赖于同一数据在不同时间的值，那么事实上它们是依赖了某个（相同的、底层的）数据层次并且 A 和 B 存在时序的逻辑依赖。

图 4-38　层间依赖：对数据时序依赖关系的分析与解构

接下来我们讨论第二种时序依赖关系。如果 A 和 B 间存在相互的逻辑时序依赖关系，那么我们总是可以通过添加一层数据抽象，来将向上的逻辑时序依赖变成“数据的”时序依赖(44)。由此将只剩下一个向下的逻辑时序依赖（并添加了两个向下的数据依赖），如图 4-39 所示。

图 4-39　层间依赖：对逻辑时序依赖关系的分析与解构

可见这两种情况下产生的向上依赖 mx 都是可以被消化掉的。并且，对于图 4-39b 中的逻辑依赖 m1，依然可以通过再添加抽象数据层次来变成数据依赖，如图 4-40a 所示。并且这样一来，我们会看到一个结果：A 与 B 之间不再有依赖关系，且 Z 与 Zn 之间也不再有依赖关

系。那对于这些并列的层次/域，也是可以归并在同一个层次之中的（如图 4-40b 所示）。

图 4-40　层间依赖：归并

由此得出的进一步的推论是，上层对下层“可以”仅有确定的数据依赖关系：下层对上层的调用，总是可以通过数据的下沉来避免；多层之间向下的调用，可以通过公共数据层来避免。这是一种理想状况(45)，也是“逻辑－数据层”这种两层基本结构的由来。

然而还有最后一种情况，会对这种理想化的层次结构设计造成一定的冲击。这种情况发生在将嵌套结构转化为层次结构时，其“核心部分”必然位于底层——因为它是可确定的、可预先确知的。如此一来，在这一层中必然存在底层向上层的关系，如图 4-41a 所示。

图 4-41　层间依赖：对嵌套结构转化为层次结构的处理

但这个关系 m2 究竟是什么关系呢？如果我们将 D..Z 都视为逻辑，而 A'（作为包含内部关系的核心过程）就必然会发生 A'到 D_Z 的逻辑依赖。但是，我们也可以将 D..Z 都视为数据(46)，这也就意味着 A'依赖于“未确定的或动态增减的数据”。这是容许的，因为这只需要一个注册过程，以便将数据动态地置放到底层即可，如图 4-41b 所示。

这样一来，向上的逻辑依赖 m2 就变成了 A'对 Z（动态数据）的、向下的数据依赖 m0。对于 A'来说，Z（注册逻辑）是确定的，并且 A..C 与 Z(注册逻辑)之间的关系确定，因此 A'的整体也是确定的(47)。

六

通过讨论领域间的组成与关系，我们可以尽量将系统的可变性隔离在较晚实现的域中。因此，这意味着先期建设的系统总是不变的、稳定的、可重用的。这是组成论视角对系统架构的主要贡献。但从系统演进的趋势来说，任何系统的组成部分都必将面临我们持续开发行为所能带来的影响。

而界面（interface）——就提出这一概念的本意来说——就是通过对系统确定性加以规格化，从而来避免上述影响(48)。在我看来，如果一个界面（以及其规格细节）是确切而有效的，那么它应当完全满足如下条件：

□　准确——合适的知识与表达，至少能让交流双方通过某种形式沟通；

□　有用——完全明白的意图，至少与系统架构的意图不违背；

□　可见——执行的效果显而易见，至少在领域或层次上的数据与逻辑流向明确；

□ 可能——应当存在实现的手段，至少可以立即着手开始尝试。

在架构的表达上，由于纵向分开的并列部分之间是没有关系的，因此它们之间也就没有规格化的需求。而横向的层次之间，仅有向下依赖是确定的，因此界面必是由下层来规格化的。这应当包括（上层所需的）数据规格与（调用的）逻辑规格两个部分。

究竟是“上层所需”还是“下层具有”决定了规格的细节呢？这是一个相当关键且又颇具争议的问题。从此前的讨论来看，上层总是（在时序上、相对的）还未能确定的，因此依据“上层所需”来决定规格细节显然是缘木求鱼的事情。但反过来说，如果仅凭“下层具有”来确定规格细节，虽然在逻辑上讲得通顺，却又常常因为对这些规格的细节考虑得不够周全而限制了上层的使用，这又是层次架构在实效性上的疑难。

有两种方法来改善这一问题(49)。第一种方法是，我们不必过早追求底层界面细节的准确性与可用性，而仅仅将底层所能提供的功能（逻辑）与数据完整（而又粗略地）表达为接口 Intf_L0_v0.1。接下来，在后续的开发活动中，上层的开发活动可以基于这些功能与数据进行设计细化，并将这些设计视为对 Intf_L0_v0.1 的封装（例如代理）来实现为 Intf_Ln_v1。然后，我们只需要将 Intf_Ln_v1 下沉到 L0 层，并以之为公共接口 Intf_L0_v1 发布即可。简单地说，就是在上层细化接口并交由下层发布。尽管这看起来颇为复杂，但确实是实践中常用的方法。

下面讲第二种方法，参考我们此前的讨论，向下依赖其实只发生于如下两种情况：

□ 上层对下层的数据依赖(50)；

□ 当采用嵌套结构时，需要一个向下的注册逻辑。

那么，我们其实是在说：一般层次系统只需要数据界面，而框架/引擎类的层次系统只需要多维护一个注册逻辑即可。所以 REST 和 CRUD(51)事实上成了应付大多数情况的抽象接口方法；即使我们是实现引擎或框架，也只是需要在核心层面考虑清楚注册与回调的机制。

当然，采用第二种方法将意味着上层总是（尽可能多地）在面向数据开发，而非面向既已确定的逻辑，所以这样建立的系统将趋于扁平(52)。这样的架构在应付系统整体规模与复杂性上的能力其实是不够的。因此采用第二种方法常常也是权宜之计，在一段时间之后，仍然会从上层中抽取确定的逻辑来形成新的层次(53)。

但我们如何确定“这是一个界面”或“这些既有的东西可以表达为一个界面”呢？

对于架构中的一条横向的线来说，确定它的界面设计的原则有很多。大致地，则可以分为系统、表现、模块三类原则。其中“系统原则”是总的纲要，“模块原则”是系统自身进行（已经进行和计划进行）层次或领域规划时的指导，而“表现原则”是界面的风格与样式方面的约束。表 4-5 列举并分类 Erlang 的一些实践性原则(54)。

表 4-5　三类界面原则的示例：Erlang 的一些实践性原则

* 写出朴实无华的代码

** 不要在全局声明和使用私有数据结构，不要暴露对象的实现

*** “防御”与否是一个有争议的话题

最后，仅就设计的表达来说(55)，也可以将一些 GoF 模式作为确定的、确实有效的界面设计参考。一般来说，其结构型模式适合数据层次的规划，而行为型模式适合逻辑层次的规划。后者，尤其是在实现嵌套结构的层次化中相当有效。

————————————————————————

(1) 这的确是一种好的开发模式，它确实减少了架构决策过程的复杂性。

(2) 这里的意思是说，系统是我们的目标，项目是围绕这个目标提出的解决方案。作为解决方案，意味着它包括技术、产品、团队、资源等所有有关“实现该系统”所需的方面。而架构是整个解决方案中与技术相关的一个部分。仅从规模（的领域相关性）上来看，我们可以把整个 Facebook，或整个 Google 网站作为“系统”的一个参考对象。

(3) 其中，某些（与计算机无关的）领域细节是不需要在这里讨论的。

(4) 如果架构是一个“决策”的过程的话，多样性就是必须的。

(5) 在这里所言的“独特认识”是认识方法论决定的，而不是对象本身的特点所决定的。因此领悟能力是跨架构（或说超越架构）的。脱离具体的架构对象，是我将这一能力称为“领悟”的原因。

(6) 例如，说某人是行业领袖，并不意味着他在行业中担任了类似于联盟主席之类的领导职务，也并不意味着他是行业对口的政府职能角色。行业领袖只表明了他在这个领域中的突出表现以及影响力。

(7) 参考本书总论：其一，架构是目标之于规模与细节上的投影，因而其内在就是跨项目组织的管理与实现两个轴向的；其二，架构本身是通过意图来保持与经营者（在领域性、方向性与战略假设等方面）的一致性，因此其外在也是经营者施于系统的影响的一部分——这本质上也是跨组织结构的。

(8) 分工并不是规模问题。例如，数据架构、前端架构这样的分类法是指分工，而下面讨论的技术架构、系统架构等是指规模。但分工与规模的组合，大致说明了一名架构师在团队中的位置。例如，前端平台架构师，或运维技术架构师。

(9) 仅软件开发领域而言，架构师的领域能力与他能应对的开发规模是相关的。参考第二篇中提及的四种开发规模：技术架构对应于功能（function)与程序（program），应用架构对应于应用（application），而系统 / 平台架构对应于系统（system）。

(10) 现实中我们常常会提及“平台架构师”这样的职务称谓，事实上它是系统架构这一规模下的具体描述，因为平台/平台化是系统的一种实现。因此准确地说，应当将之统一称为系统架构。

(11) 参考总论的 VEO 模型。

(12) 首席架构师在某一具体领域可能并非最强，这是因为首席架构师所应对的首先是“架构”这一领域，其次才是“目标系统”的具体领域。

(13) 一种可计划的问题解决模型。通过“制定、检验、重构”过程的反复迭代，逐渐拟合问题表征与实施计划，以最终解决问题。参见《认知(第二版)》，Glass 与 Holyoak 著。

(14) 将架构本身作为一个领域，则这些知识是为架构而形成的，与原始的系统目标（例如产品特性等）无关。

(15) 注意决策能力、领袖能力与管理能力并不同一，这里只讨论其中的决策能力部分。而将“架构团队”作为一个组织来看，也存在对“管理者”的需求。但即便如此，也并不表明首席架构师必须担负架构团队的管理责任。

(16) 本书只讨论了决策过程模型，但并不否认团队推动架构的其他过程方法。

(17) 一般意义的“功能架构”是“实现架构”中相当重要的组成部分，例如对客户需求项的映射，但这未能在图 4-18 中体现。

(18) 接下来两个小节，是对“架构形成模型 M0”的一个应用。我们假定了一个实例，并基于此实例展开了一个系统架构过程，注意它并非是一个已经被理论化的架构方法，因此许多名词借用了其他场景。

(19) 从现在开始，你可以想象成有一个“首席架构师”决策了这一系统架构模型，而我们后续要讨论的是其他架构的形成与表达。我们再晚一些会谈到“这一决策”事实上是与架构师自身的经验、背景有关的。

(20) 架构设计不单单依赖模型，也不单单“产出”模型图。架构过程在最终的架构文档中应当包括模型图、规格文档，还可能会包括关键子系统的形式化代码与流程图。在不同领域的架构中还包括特定的模型图（如数据架构或部署架构图），这一类的架构过程可能会将一部分设计工作也包括在内。对于我们在这里讨论的“架构”行为而言，仅仅是指架构师通过

模型或抽象分析来得到系统的基本映像，并将它表达为一些图例以用于后续架构过程的指导与分析即可。

(21) 一般会把“process view”（参考 Philippe Kruchten 的“4+1 视图”）所对应的架构称为运行架构。这里采用了这一名词，但含义上有些不同。在本书的这一示例中，运行架构更确切地说是“系统运行环境的架构”，它表达了在系统架构的层面上对开发的限制与约束，因而包含了对服务和服务的运行框架的描述——这里的“服务”是一个与“结点”相对应的概念，而并非一个确切的（如 Java Runtime 中的.jar 包）包或组件。

(22) 在架构实作中是不应当存在所谓“暗示”的，架构师应当明确地将自己的意图表达出来。不过，“明确”的手段就未必是模型图了，因为架构师可以选择更为详实的架构文档来陈述这些意图。在这里，我们只粗略地讨论这种意图，并且为最终的（在本小节结束时将提出的）架构意图而设下伏笔。

(23) 因此，其一，基于对我们当前的、现实工作的过程回顾，是不可能找到“获得架构意图”的一般方法的；其二，我们在上一节中并没有再现某种实施过程，而是讨论了这一过程中的（可能的）思想脉络。

(24) 换个简单而直白的说法，“架构是做出来的”这一观点就是这一架构方法的基本论调。

(25) 注意，我们讨论的系统是“领域集”，因此这两个解对于“领域集所对应的行业、行业链”也是有效的。例如将语境置于：我们要构建这个行业生态链下的核心基础平台。——不过，事实上我并不知道我在说什么。

(26) 架构为什么要分层以及如何理解架构的分层，是下一节讨论的内容。这里所谓的三层架构或多层架构这样的模型，是讨论中用以参考的、现实中的例子而已。

(27) 既然我们在考虑“战略”问题，那么就必然有更多的角色会影响“办公系统”的架构过程。

(28) 本例参考了《微计算机应用》2003.02 期，“基于工作流的 OA-ERP 集成”一文，作者郭应中等。

(29) 《人月神话》对这一问题也有过充分的讨论，其基本观点是“由于精确性的考虑，我们需要形式化的设计定义，同样，我们需要记叙性定义来加深理解”。

(30) 它表明的其实是“普遍/特殊”这样的分类法。

(31) 领域与层次的分类依据是架构中非常重要的信息。一般有三种方法来表达它们，其一，通过线条和标注；其二，通过更为明确的分类指称；其三，在其他架构文档中加以详述。

(32) 在“第二节　系统的架构与决策”中，是把规模作为复杂性的一部分，讨论系统在“领域集总量”上的规模。

(33) 在本小节有关复杂性的叙述中，“1”是没有单位的，表明它的确定性；当它与“（计量）单位”同时出现时，才能表明复杂性的大小。本书不讨论“如何计量系统复杂度”的问题（亦即是说不讨论单位的设定），仅以这种抽象描述的细微差异来说明“复杂性的大小与可变”之间的是存有区别的。

(34) 类似这种“性（状）”的不变，并列与嵌套结构表达的分别是系统的“（总）量”与“（本）质”上的不变。

(35) 就规模性而言，本书是将“库”划为应用（application）这个级别的。这里只是对它的构建与表示方法加以讨论，并不是否定此前的规模划分。从另一个角度上看，泛义的“系统”也是可以包括库、程序或功能的，因此其构建与表示方法也有可借鉴之处。

(36) 你可以将某些引擎（Engine）也视为框架，它是符合这里的讨论的。另外需要补充的是，这里的“框架与平台”与上一节小中的概念是相同的，这两个小节分别讨论到他们的范围（意图、方向与目的）与结构。

(37) 例如，引擎层与处理层、驱动层与应用层、框架调度层与业务层等，前者都是核心领域或包含核心过程。

(38) 可以理解为：向系统追加逻辑，而它们不依赖底层系统时，这些新的逻辑只对系统规模构成影响，而不影响原有系统整体的稳定性——例如我们可以将新的逻辑独立于原系统部署。

(39) 进而也会破坏由确定性带来的稳定性，持续性等架构特性。

(40) 我们讨论的是“如果确实具有这样的关系”，那么如何在层次中予以处理的问题，而非鼓励这样的关系。

(41) 注意我们并没有讨论“同时发生相互依赖”的情况，若 A 和 B 间不存在时序性，则似乎它们并不应当被拆分。以咬合的齿轮为例，若分离二者则齿轮之间的互动性全无；仅当视它们是一体时，才会存在“同时相互依赖”的逻辑。

(42) 由于向下的关系是系统中既存的，因此不必讨论从 B 抽离至 A 的情况。

(43) 方案的选择取决于成本，例如考虑网络开销或数据重构与存储的开销。

(44) 参见第八章第二节。

(45) 绝对的数据与逻辑分离可能导致数据规划、迁移和转送的成本失控。同时，也会导致逻辑间需要维护更多的状态或消息，这一定程度上增大了复杂性（尽管仍然是确定的）。因此，在现实的多个层次之间，通常是既允许向下的数据依赖，也允许向下的逻辑依赖的。

(46) 注意，这里是将它们“视作数据”，而非“通过抽取数据层来消除逻辑依赖”。所谓“视作数据”，是抽象概念上的重新定义，请参阅第四章第三节。

(47) 事实上还必须讨论 A..C 与动态数据之间的关系，这取决于“引擎/框架”等具体方案。一般来说，这一关系也是确定的，例如调度关系，又例如设计模式中的策略、命令等。除了 Z（注册逻辑）之外，在对动态数据 D..Z 的使用上，还应当考虑安全性、稳定性等因素，但这些都取决于“框架”的详细设计，并非这里讨论的架构层次规划的问题。

(48) 在 Walter F. Tichy 在 ICSE1992 上的论文“Programming-in-the-Large: Past, Present, and Future”中提及了层次系统与其界面抽象的出处。其译文“大型程序设计的过去、现在和将来”发表于《计算机科学》1992.06 期，译者陈海东。其原文为：“层次模型中体现的数据抽象原理可追溯到 1966 年 Dennis 和 Van Horn 的论文，它强调了用户和内核间的一个简单接口。Dijkstra 在 1968 年报告了第一个可供使用的、内核分为几层的操作系统。”

(49) 界面的设计是自下而上的，这是层次结构的形成和表达等内在性质所决定的，但其便利性则取决于设计者的经验。所以这里必须强调的是，这些仅仅是依据经验来讨论的一些技巧，并且也仅能予这一局面以有限的改善而已。

(50) 逻辑依赖可以通过数据下沉或添加数据抽象层次来变成数据依赖。

(51) REST（Representational State Transfer，表述性状态转移）是一种面向远程服务提供的架构方法，它将架构中“端到端”的关系理解为“资源需求”，并将主要接口抽象为 GET、POST、PUT 和 DELETE 四种方法。而 CRUD 则抽象了面向存储/持久层/数据的四种基础操作：Create、Retrieve（Query/Select/Read）、Update 与 Delete。

(52) 这里的意思是说层次过少，尤其是具有确定逻辑的层次过少。

(53) 平台是从下向上做，还是从上往下做？这个问题的答案其实并不绝对。一般来说，我们能根据经验来确定大体的层次，并在各层的细化中采用“上层实现，下层发布”或“从上层的确定逻辑中抽取层次”的方法。总的来说，这一过程是渐进的，而非一开始就决定的。

(54) 引自“Sofrware(SW) Engineering Principles”，出自 Erlang 项目的公开文档“Erlang Programming Rules”。

(55) 本书并不详细讨论“架构的设计”或从需求开始的“分析与设计”过程，而是非常严格地区分架构过程与设计过程。因此架构表达与设计表达并不是相同的意思，前者的目标是从系统中识别出的基本模型，而后者则是在该模型上的细节刻划，因此后者是可以进一步地借助 GoF 与 UML 这样的工具。注意，尽管 UML 图也分结构型与行为型，但这与我们讨论的内容并不“完全对等”。

第十二章

架构原则，技艺、艺术与美

我对架构的认识与思想，只是架构可能的认识与思想中的一个方面，是其可能的解中的方式之一。我必须提及这一方面与方式的核心指导原则，这些原则的正确性必将表达为：它可以为其他的认识与思想提供依据，是其他有效的、可供讨论的认识与思想不可违逆的基本前设。本书对架构的谈论，只是这些原则下的一个运用示例；这些原则来源于这些示例的思考过程，并超越于这个过程的结果——本书所讨论的架构。

平衡是一种技法，进而也是一种能力。与此相同，眼光也是一种能力。区别是眼光的能力不仅在于点滴积累以获得经验性的娴熟，也在于对事物本质的拷问。所谓眼光的不同，不仅仅是我们发现问题与解决问题的具体方式之异同，更深一层则在于我们的思想之异同。例如，眼光可以发现大象之巨，平衡可以处理称象以微，而思想则在于反反复复地拷问：象之巨与秤之微的冲突本质是什么？本质是象与秤的关系吗？这个本质问题的解是什么？

或者我们可以前行一步：这个本质问题的解的含义仅是计算或求值吗？又或者，我们回溯至第一个问题：我们发现的本质，是本质吗？

这是一个死结。思想的起点与终点都在一个循环之中，故而无始无终。

第一节　架构原则

一

架构第一原则：架构面向问题，但满足需求。

1. 我们已接受的许多东西是有着商业背景的

事实上，我们已经被迫接受了许多种“面向些什么”的架构实现，这既包括“面向企业的架构”这样直言不讳的，也包括像“面向互联网络解决方案”这样披着外衣的。总而言之，这些架构的推广者总是试图声称它们在某个方面具有丰富的经验，有着广泛而显著的成功案例。并且，他们试图用种种措辞来掩饰“这些架构”的背景，而寄期望于你去复制这些结果。

这种“复制”的背后就是商业行为了。

然而几乎没有人会走过来深入你的系统，帮助你指出你的问题在哪里，以及需要如何应对它们或者绕过这些问题。因为如果你的系统没有了问题，那么看起来也就不需要解决方案了，那么看起来方案提供商也就没什么事可做了。

事实上，我是反对“方案”这样的东西的，因为，在大多数情况下，那些向你推广这些东西的人根本就不了解你的问题，他们只是通过对你的需求的了解，拼凑了这样一套方案以应付你那些急迫的购买冲动罢了。

2. 面向需求通常是不考虑系统的背景的

从提供方来考虑的方案，通常是面向“同类系统的同类需求”的。这种需求上的相似性才决定了方案的价值。它并不考虑确定系统的背景，因为背景的不同正好削弱了方案的价值。然而，我们事实上是无法脱离背景来讨论系统问题的。举例来说，如果问题是“某个模块导致了性能较低”，方案是选择某个.NET 架构方案，因为它已经被证实过在这方面异常优秀。那么选择正确吗？

虽然看起来我们解决了“问题”——性能较低，但事实上这并不一定是正确的架构选择。因为如果开发方、用户方根本没有.NET 技术人员，则这一选择事实上没有解决问题，反而制造了更多的问题。又假设我们确实有很多.NET 技术人员，但是系统当前并没有围绕.NET 架构方案来实施，那么这一选择就增加了问题的规模。所以问题本身是有背景的，而它的需求可能表现出来与这一背景无关。

3. 面向问题首先是客户视角的变化

“面向需求”本身是没什么错的，因为我们的软件开发活动最终总是要解决用户的实际需求。但需求的“持续可变”是所有问题浮在冰海上的表象，正是它们随海水的、风力的变化而变化着，才导致我们“面向需求”去求解时疲于奔命。这其中，一个重要的问题在于：客户是很难从系统角度上识别问题的，并且当他们站在“客户与供应商”的层面上思考时，他们也完全不必要对可能的系统问题作出解释。

提出需求，这近乎于客户的本能，否则他们便不需要供应商了。但对于问题，却只有当客户将供应商视作“合作者”时才可能提出来。从“采购－供应”的视角上看问题，客户与供应商是争利的，只有站到“共同解决问题”的角度上来看，二者才是共赢的。传统的工程方法以及架构、开发的思路，事实上已经主动将客户摆在了对立面。因此，出现“你们——作为客户必须在需求文档上签字”这一局面就是必然的、顺理成章的事情。而客户与供应商“共同”面向问题时，问题本身才是焦点：需求可以通过对问题的阶段性关注、梳理来明确；需求的变化可以通过架构的确定性来消化。

退一步来说，若既已存在“客户与供应商”这样的事实关系，那么供应商从面向需求转而面向问题，仍然可以最大程度地得到客户的谅解——尽管“面向什么”在客户眼里确实是一个过于空泛的概念（这也是尤其要注意的地方）。对于“供应商/开发方”来说，面向问题会是一个主动发起合作，进而争取普遍合作的开端。这无论对内部项目、自有产品还是外部项目来说，都有着明显的积极意义。

4. 面向问题与开发实作并无冲突

但是“面向问题”这一概念对于开发人员同样显得空乏。因为问题的关键求解在于架构，而

不在于具体实作阶段的某一个技术行为。以平台层次为例，假定架构对某一问题的决策是“数据建模的过程可以延后至第二阶段”，这意味着平台层次中的底层数据结构是模糊的。那么这时开发人员如何做实施呢？答案是：开发人员可以在任意时候、任意位置，就地实现数据库或数据结构(1)。但是，这必将给架构角色带来层次规划上的灾难。因为如果推进这一方法，则在“第二阶段”来考虑数据建模时，系统架构将无法进行调整以容纳、应用新的数据模型。

因此，架构在第一阶段既不能“放任”开发人员的数据规划行为，也没有足够的信息与时间来进行数据建模。但这一矛盾的实质并不在于“谁做数据建模”，而在于“何时定义其细节”。而使架构角色在这里陷入了两难困境的原因则在于，他对自身的职责仍然缺乏必要的了解。回顾此前我们在架构过程中提及的两项架构责任：

□ 其一，架构对实施的约束；

□ 其二，架构的阶段抽象在实现域与交付域的映射。

由此看来，架构应当在第一阶段中与开发人员约定（注意做这些约定，其本质上也是数据建模活动）：

□ 开发人员的数据规划行为必须限于当前应用中的数据层；

□ 必须通过一个界面交付到应用层，避免直接访问；

□ 若该数据规划涉及多个应用，必须由架构角色来确认规划的有效性；

□ 数据层的交付界面必须不涉及特定数据层实现方案的细节(2)。

这些约束将对实现域中的行为构成明确的影响，并且也将影响到部署域。这些影响使得开发人员无法透过“自由地数据规划”来直接影响第二阶段中的数据建模工作，也不会对各个阶段中的部署过程构成威胁。

但是反观上述约定，其事实上也不会对开发人员的具体实施造成“巨大”的影响。架构的约束既体现为对问题的把握，也体现为面向问题的、阶段性的隔离。它对整个系统工程构成影响的方式既包括一系列架构图例，也包括上述的一些实施规则，最后——也最为重要的是，还包括架构师对问题的分解。

5. 面向问题是架构活动的必须

软件架构活动的来处并不在于“变化的需求”，只有将架构所解决的本质对象定义为“问题”，架构本身才有长期与持续性的需求；架构本身的复杂性与规模才有出处；架构应对于“持续可变的需求”才能寻得方法。

总的来说，需求可能一样，但问题却未必相同；需求可能被满足，但问题未必会因满足需求而消失；需求可能是破碎的，但问题却恒久而弥新。因此，架构的思维对象必须直接指向问

题。唯只如此，架构活动的本质，才在于面向问题的求解；而其结果，才会是一个长期的、有效的、可持续推进的架构，而非应对一时之所需的技法。

二

架构第二原则：架构基于概念抽象，而非想象。

1. 形式化方法

作为第一原则，“架构面向问题”是无助于讨论“架构是什么”这一设问的。架构作为一个确定的工作产物，它必须有对其形态的确切说明，否则我们无法以之作为后续实施的依据。举例来说，若“架构师所想”是架构，那么架构的本意是无形的，它在被叙述的一刹那便已走了模样；若“架构师所言”是架构，那么架构最终必以录音为载体，并且后续的分析也将基于对录音的讨论。类似的，我们讨论架构的形态，是要讨论架构本身可否用作持续依赖（我的意思是实施）和持续讨论（我的意思是不同阶段的架构），并更具体地阐明“依赖与讨论”的可行方法。

不幸的是，总体来说，在这个问题上我们的可选答案并不多。就目前对思维表达方法的研究来看，我们只有意象化和形式化两条路可走。意象化包含联想与想象，例如说作者 A 在纸上画下一个圆，观者 B 可以自由地认为那是一张面饼，或者是昨晚所见的月亮。至于这一意象是否确实是 A 所绘的这个圆的本意，是不要紧的。如果非得说这一意象有传递的效果，那么我们可以强调 A 绘制的圆表达了“完整”，而 B 所见的面饼与月亮总的来说在形态上也是完整而无有或缺的。

自然语言确实是形式化的，例如我们可以强调主谓宾这样的结构。如果言者 A 在表达的时候只满足了谓宾结构，我们还可以进一步地细化规则，来讨论“主语”在语法上的承前省略和蒙后省略等问题。但总的来说，自然语言一方面是形式化的，另一方面确实有着相当的随意性(3)。例如，我们在此前讨论过的“两岸猿声啼不住”，其猿声是否出自真猿的问题，这一问题在自然语言的形式化语境下就是无解的。

从非形式化到形式化，一路走来，我们唯一可选的是“更加明确的形式化”。这是表达架构——这一思维活动的结果的最终方法。

2. 形式化的基础是抽象

但是形式化本身只是一个方法，我们不能说“用包饺子的方法包出来的就一定是饺子”，这是方法之于事实的区别。我们必须将“架构是什么”最终确指到事实，而不是简单地定义它的实现方法——我们必须时时反省：形式化方法本质上只是“在我们现在、在对思维的表达方式过于粗略的前提下的、不得已而为之的”一种方法(4)。

形式化到底要表达什么？是什么决定了形式化作为一种方法的有效性？回顾这两个问题，其核心在于：其一，在表达之前的思维活动中，究竟形成了什么；其二，在表达之后的验证活动中，我们可选择何种方法。前者必当我们于头脑中形成一个确定的事物，才能将其形式化地表达出来。这一事物，必是抽象的，而非具象的。关于抽象与具象的问题，我们已经反复

论述过了。这里只强调一点，具象是可以表达的，例如图画或雕塑，乃至于音乐；但具象的表达是基于感官上的一致性的，而非基于——像抽象那样——在概念上的一致性的。为了说明这一细微措辞上的区别，我们可以假定回到原始社会中，原始人 A 向 B 举出一根树枝，假设他要表达的是昨天他所见的另一根树枝，那么这是具象的，二者不同一但是等义；若他要表达的是“1”个某种东西，那么这是抽象的，二者——一根树枝的“1”，与一个某种东西的“1”——是同一且等义的。

抽象的问题在于“1”必须是原始人 A 和 B 都能共同接受的一个概念。若这个概念原本不存在，或无法通过其他概念或方法予以传送，那么 B 将无法理解 A 的这一形式化的含义。这也是佛陀拈花无解的原因，因为这一形式不存在任何概念，也不存在让弟子们理解这一形式的、其他的概念基础。

确定的形式必然包括抽象、概念以及基于此的确定表达法。否则它必将无法作为我们表达确定思维的基础构件——与此相对应的，意象适合表达的是非确定的思维。我们希望构建“系统”，并对该系统作出引导它中长期发展的“架构”，因此这一思维的结果应作为一个确定抽象，并以确定概念来陈述。若是不确定的，那么我们无法正确表达给原始人 B——当然，也无法表达给某个程序员。

抽象是不具体的，但抽象的表达是确定的；具象是确实的，但基于具象的表达却是不确定的。如上二者互成矛盾，但是却构成我们思维与表达的全部极限。作为架构的目的——产生确定的系统——的所需，我们只能选择抽象。而所谓形式化，只是“思维的抽象表达”的一种方法(5)。

3. 形式化的表达必须以语法和语义为基础，而忽略语用

架构存在的基本价值在于交流，如果不需要交流（例如只有一个开发人员的个体工程，且该开发人员总能自始至终地(6)明确“在做的软件”与“事实的系统”之间的映射关系），那么这个开发活动中就自然不需要一个“具形的、存在的架构”。

总的来说，交流有两个基本的要素，其一是交流的主客体(7)，其二是交流的对象。例如，我们说“世界各国的经济文化交流”，那么总是包含上述两个要素的，其中的交流对象就是“经济文化”。又例如，我们说“集装箱促进了全球货物的交流”，那么对象就是“货物”。问题在于，这一类的“交流的对象”总是确实之物——无论是虚的经济文化还是实的货物，那么它需要的是一个载体。然而，另一类的交流所指的对象却并非“某物”，而是“某物的含义”，这种情况下，我们的可选工具——而非简单含义上的载体——只剩下了语言。这里说“只剩下”，是因为在我们人类的抽象概念中，只有语言既包括语法又包括语义，并且使用的是语法来交流，而“交流的对象”却是其语义。

任何有语法与语义并以语法为交流形式，以语义为交流对象的，都可以称为（广义上的）语言。举例来说，眼神交流是伴随着形式的，我们所谓的眼神不单单是指瞳仁，还有瞳孔的大小以及眼皮的张开程度等——若抛弃这些，我们是达不成眼神交流的。但就眼神的形式而言，瞳仁发红是病症，瞳孔发散是死相，眼皮张开那是醒着。这些形式若不包括某种语义——我的意思是愤怒——那么它只能表达一种医学的或生理学上的现象而已。

所以若将眼神视为语言交流，也必是有语法和语义的，而且必然表达为语义的交流——后者才是它作为语言这一概念的充要条件。再回到我们的形式化的问题上来，我们尽可以有任意多种形式，也包括这一形式的要件（我是指概念、抽象与表达法），但如果要表达架构师的思维，那么它还必须以语义为交流的对象。这是“架构师应以形式化语言为交流工具”的一个推理过程，在“形式化”上，它是指语言工具的基础要件；在“语言”上，它是强调语言的语义特性。

“忽略语用”仍然是考虑“架构的目的——产生确定的系统”的所需，而进行的一个选择。这一选择事实上仍有争议。比如说，架构师确实会在实施中将一幅架构图用于不同环境下、应对于不同对象的解释。就语言上来理解，该架构图表达的语义便因语用（该语言的用所）的不同而不同。但这带来了一个问题，也就是：架构师所表达的系统是不确定的，在交流客体的感受上会变成“架构师主观而随意地阐述着系统”。

所以忽略语用是一种选择而非一种必需。基本上来说，如果 A 与 B 有足够的、相当的架构能力——因此他们能基于相同的背景做出相同的理解，那么二者之间仍然可以随时切换着语用环境来交流“同一幅架构图的不同含义”(8)。但如果将这一行为扩散到整个项目团队，那么既是不公平的，也是不可取的。

三

架构第三原则：架构=范围+联接件。

1. 基本前设

架构的目标究竟是什么？我们当然知道其目标是系统——无论是大的、复杂的体系，还是一个小的、有含义的组成，又或是我们要考虑其系统性的任何东西。然而这一概念下的系统，其内涵是丰富以至于无可穷尽的。架构作为一个事实工具或对于这一系统的事实影射，只能表达其中的部分而决非全集。因此，我们所谓“架构的目标是什么”，其答案必将指向系统，也必然是系统特定的一面两面或数个方面，这是我们在这一前设中必须明确的。

对于本书的总论与上一章中所讨论的架构，在总论中已经确定地将它的源起指向“系统方向的必要性”。若某种架构并不以“系统方向”为目标，那么它不适宜作为这些章节要讨论的基本对象，亦即它们在基本的抽象概念上是不同的(9)。“架构=范围+联接件”这样的求解是特指面向系统的方向问题的。若是讨论系统的其他问题，则相关的求解仍可以称为“架构”，并仍满足第一和第二原则，但未必满足第三原则。

本原则是对第二原则的补充，讨论架构作为工作产物时的内容。

2. 范围与联接件之于系统的意义

决策层在系统的方向问题上赋予架构师的职责是“目标的映射”。这包括两方面的含义，其一，不一定是确实的目标，例如某个产品或产品的某个版本；其二，是对目标的约束，而非说明其实施的细节。范围与联接件是架构师的两个工具，与其说它们是对规模与复杂性的求解，不如说它们事实上就是架构师对“系统的方向问题”的两个求解。

所谓方向与目标有一些基本性质，包括：其一，系统的方向可能是确实的，也可能是阶段性变化的；其二，阶段目标清晰而明确，但方向却可能存有模糊性；其三，方向必是一个面的问题，而目标方才是点的问题。架构的很大一部分工作，便在于把握这些“模糊的阶段状态（阶段目标或产品版本）”背后的系统关键，通过联接件来刻画系统的脉络。无论系统在中长期上的变化为何，这些通过联接件得到的系统脉络是很难有变化的。比如说，我们很难改变 Web 中用户行为的流向，总是从主页到二级、三级或更多级页面。但是，当某种行为模式提出来的时候，这种脉络就变化了。例如，以用户为中心的 SNS 网站，那么就会是从登录/验证开始，并经由不同用户行为引导而形成流向。那么这两类网站的联接件与联接关系就会非常不同，而各自的（关于联接的）架构也必是在这一网站的发展过程中长期不变的(10)。因此，相对于系统的多个阶段目标，联接件（及其联接关系）总是纵贯其间的。

但是联接件只是解构系统复杂性的一个手法。如同我们在层次架构中通过“逐层清晰”来解构系统复杂性一样，这一手法通常用来确保系统长期的不变性——复杂性通常是由可变性引起的。

架构在应对系统方向下的规模问题时，采用的方法通常有两个：其一是对“系统组成”的明确约定，例如模块图或（细化的）层次架构图；其二是对系统构件的明确概念。后者——构建明确概念是架构抽象中最困难而又最重要的工作之一。例如，我们在讨论办公系统时，对于“办公业务系统”与“办公管理系统”的概念定义与辨析。当架构师使用这样一个词汇（定义，或称之为“概念”）来表述目标系统时，事实上就是对系统范围的明确约束。但问题通常不在于如何表述，而在于架构师“何以确定这一表述是符合系统方向的”(11)。

系统总在变大(12)，在它的形态与内涵两个方面都必将存在失控的风险。这两个风险是骈生的。此外，风险与机会也是骈生的，所以在架构上不单能够反映系统的“范围与联接件”，也可以反映系统的“转折点”。只是后者常常仅被视作风险而遭到严防死守罢了。无论从哪一个角度入手，范围与联接件都是架构用于保障系统方向以及提供系统在方向上应变的可能性的主要工具，架构所表达的是此二者的全集，而非其一或其他。

四

架构第四原则：过程之于结果，并没有必然性。

1. 基本前设

所谓工程，是一个实作问题，简而言之，工程讨论的就是如何把东西做出来。在这个问题上，架构工程与软件工程类似，也是可以追溯到“过程、方法、工具”三个要素的。其中，架构第二原则主要讨论的是方法论问题，间或讨论到与方法论适配的工具问题；架构第三原则可以视为对工程产物的补充。

形成论与组成论是两个过程观点，前者是过程论的动态模型，后者则是静态模型。将架构结果作为工程产物时，静态模型强调架构的构件之间的结构关系，以及通过这些结构关系来维护“架构目标的系统性”的方法；动态模型则强调架构是一个与时间相关的产生过程(13)，在时间轴以及组织性上，架构团队以及系统的参与者都是变化的，（整体来看，）其结果在形

态上也是变化的。在后者——形成论的视角下，架构结果是可以通过阶段进化来获得的，而至于这一产生过程是否是一次性的或迭代数次的，则是过程实施中的选择。

与第三原则一样，本原则也是对第二原则的补充，讨论架构在正确性上的一般逻辑。

2. 有关过程正确与结果正确的讨论

“正确的步骤会产生正确的结果”是丰田模式(14)的核心原则的重要组成部分。这一原则有其逻辑上的论证：设正确为 1，错误为 0，则一个结果的正确性应当依赖于（其中的“+”应记为逻辑 and）：

而不应依赖于（其中“⊕”应记为逻辑 xor，“!”应记为逻辑 not）：

前者意味着“只有所有步骤都正确，结果才是正确的”，后者意味着“可以通过对错误的逆向补偿来得到正确结果”。丰田模式对后一种模式是持否定态度的，但是与此反例的是，丰田模式事实上是通过后一种模式来构筑前一种模式的，亦即是说，“修正错误”是获得一个正确的过程所必需的步骤。

传统生产过程的特殊性意味着上述论证中的（0⊕!0）应当存在于同一个过程中，例如，“铸件尺寸过大”与“打磨铸件”应当同在一次交付中发生，否则铸件必是无法用于下一个过程中的。维护这样的体系要求过程中必须对阶段产品有明确的检测依据。也就是说，“1”的有效性必须在产生“（正确的）1”的过程中被检测，检测过程与生产过程可以视为同一个阶段下的两个子过程。更进一步的，（0⊕!0）亦即“错误修正”，应处于同一个子过程中。倘若这一点不能在工序上被保证，那么实施中错误修正的时间成本取决于“差次品”影响到后续过程的时间起止点，例如在使用它之前就被并行的工序修正了。

生产过程中如果包含大量的修正过程，则其效率会变得相当低下。这是因为修正过程将使生产过程的周期变长且导致产品品质下降，这些都可以理解为是由过程的不确定性导致。因此，总的来说，尽量摒除生产过程中的修正是得到“正确的步骤”的必经之路。为了这一目标，传统的生产型企业都会有所谓的“产品研制”阶段，在这个阶段中允许大量的修正，并最终交付一个可投入生产的“正确的步骤”。“生产”作为一种工程手段，大抵上是从一个“正确的步骤”的交付开始的(15)。

但对于目前我们讨论的架构来说，生产过程还没有真正出现。我们所讨论的形成论事实上只约定了形成的阶段性，以及前启对后续阶段的影响，但未能得到阶段下自我检测的标准。阶段间的关系最终被概括为“映射”与“约束”，就目前的语言表达能力——我的意思是架构语言在表达中的准确性而言，我们尚无法通过标准与规范使得“映射与约束”可被检测，并由此形成所谓“正确的步骤”。因此我们讨论架构过程正确性的基础是目前并不存在的。总的来说，这一观点中包括三个方面的问题。其一，就个人经验而言，我认为形成论下的架构产出是过程记录而非指导性规范，但这是出于“能力上无法做到规格化”，还是出于“架构

的某些特殊性质决定了它无法被规格化”，是我仍存疑的；其二，软件系统产品通常是一次性生产的，因此它是否需要一个生产过程并将架构作为生产阶段来理解，是我存疑的；其三，即使上述两点均成立，即我们确需“基于架构的生产过程，且架构规格可作为指导性规范”，我对其可实施性（综合考虑实施成本与团队成本）也是存疑的。

但是形成论的“映射与约束”性质必将由组成论来实现。因此结合组成论与形成论，可以在一定程度上解决上面的问题。组成论主要讨论架构构件，以及基于这些构件的架构语言的表现力。严格形式化的语言是可以存在错误检测与修正机制的，比如说此前讨论的主语承前省略与蒙后省略，本质上就是这一机制在自然语言中的表现。类似地，架构语言也可以有这样的机制，例如在层次架构上可以确定“没有向上依赖”，但如果在表达中出现了向上依赖关系，也可以通过语法规则的修补来使之标准化。当我们视架构为静态的事物时，它必表达系统的一个静态的映像，其正确性的检验是针对于该映像而言的。以我们前面讨论的“架构=范围+联接件”这个求解来说，既然系统映像是静态的，则上述构成也就是可确实的、可验证的，因而这一静态的系统映像作为上述求解的一个实施，也是可以讨论其正确性的。但这一映像之于形成论，还需要有过程实施的保障(16)，而这正是在组成论视角上无法得出的(17)。所以即使我们能够通过正确的方法、过程与工具，去生产出一个正确的、组成论视角下架构，也不能以此为据来证明形成论下的方法正确性。

我们尚未能找到过程正确性之于结果的必然关系，因此“正确形成+正确组成”并不等于正确的架构。

五

架构第五原则：系统的本质，即是架构的本质。

1. 普遍性架构原则的提出

我们一贯地认为“架构是对系统的映射”，因为若非如此，我们便不需要架构。架构行为的目的就是要得到这一映像，至于其后续是基于该映像来讨论、重构或是实作，都是一个次要的、操作性的问题而非架构行为本身的目的。从这一点来说，“架构是面向问题的求解”也只是一个结果，而非完整的、准确的、概念上的架构的本义。

我们一再论及，架构只是系统一个侧面的映像；并且，我们将架构思想指向“面向系统的问题”，才进一步地确定了“（我们所讨论的）架构”是系统的哪一个方面的映像。那么，架构第一与第二原则事实上只在讨论一个狭义的架构，是“解决系统确指问题的一种架构思想与架构方法”。由此得出的推论是：我们一直在努力追寻的仍然只是系统的一个面。“架构”这一抽象在我们此前的讨论中仍是相对狭义的。举例来说，我们讨论过的“过河问题”中，若问题是“过河”，那么它的解就是“过河的架构”，其后续自然也就是做船，或者趟过去，或者游过去。总而言之，问题确定了，其解也就不言而喻。

这一误区的起源是第一和第二原则中的两个原始设定(18)：

□ 其一，架构是系统的侧象，这是就其“表象”的表达。但是，这意味着它只是架构的自我释义，是“我之我见”，作为推论系统的事实孤证是存疑的。

□ 其二，架构映射系统，这意味着系统先于架构而既存。但是，系统也许原本是不存在的。例如，问题是在某种背景下既存的——假设我们对问题背景缺乏足够的认识，因而这一背景尚未系统化——那么若基于“架构映射系统”这一观点，也就意味着我们无法进行任何有意义的架构工作。

因此，我们必须重新定义我们所讨论的架构、系统以及二者的关系，这是上述架构第一和第二原则作为普遍性架构原则的必要前提。

2. 系统性

我们已经提到过“（架构在）时间上的可持续性”，并进而推出形成论所讨论的两个问题，其一是规模，其二是通过组织过程来实现规模。但这个“可持续性”究竟是系统自身的本质问题，还是因为形成论的“所需”而带来的设问，也是“我之我见”的孤证。类似地，在组成论的视角上，我们也主要讨论了复杂性的问题。其中，在层次化的结构模型中，我们事实上是讨论了其中的一个解集，即“通过隔离可变性来解构复杂性”。

总的来说，形成论与组成论是“（面向实作问题的）过程论”下的视角。我们不能因“过程是这样需要的”，而反过来指称“一个系统的本质为何”。本质是不应随应用的需求而变化的，否则其必然是一个“可用的观察”，而非本质本身。

那么，到底什么才是“系统的本质问题”呢？

我想我能对这一问题提出的唯一可能的答案是“系统何以为系统”。也就是说，我们之所以将某个领域集或其他类似的“组成/构成/集/……”称为系统，必是因为它们之间存在某种系统性，以维持它们的内部关系与外部表现。这种系统性是系统存在的唯一依据、核心矛盾与主体价值。既如此，这种系统性也必是架构——系统所有的可能映像——的基本事实、本质问题与形成驱动(19)。

唯有将系统的本质与架构的本质都设定为对“系统何以为系统”的拷问，才能抹去二者因概念抽象而导致的差异。唯只如此，它们才能在“问题与解”上真实地一致，才能在“过程与方法”上无视于系统与架构的先后问题。

3. 本质

我们知道，我们之所以用“语言”来指代那些程序代码，是因为它们是我们与计算机交流的工具，这与我们的自然语言——在作为交流工具上的——本质是相同的。我们也知道，计算机作为物理机器能够产生运算效果是因为开关状态与二进制——在作为算数工具上的——本质是相同的。

我们已经提及过类似这一切的、最关键的、背后的假设：

在本质上相同的抽象系统，其系统解集的抽象也是本质上相同的。

综观我们的知识构成，我们所见并能自由论及的一切系统，都是事实系统的抽象系统(20)，我们只是在多个抽象系统中维持着本质上的相同。无论“问题的背景”是或不是一个既存的系统，我们的架构与这个“即将被识出的系统”其实都必将是两个“本质相同的抽象系统”。因此通过架构行为以得出一个系统，与通过一个既有系统得出它的架构，在认识论的视角下是完全无二的。

系统的本质即是架构的本质。我们必将二者的本质指向同一，其复杂性，亦即结构的本质，方可同一；其方向性，亦即目标的本质，方可同一；其系统性，亦即问题的本质，方可同一。

第二节　技艺、艺术与美

一

称象的方法是可以传授与实作的，我们称之为“技术”。就传授来说，授业者可以分解步骤、讲述原理并总结经验与诀窍；求学者可以亦步亦趋地跟随，先得其形实，再究其质底。就实作而言，实作者可以在技术的实践活动中有所变化，若这种变化是有了质的区别，我们就称之为“新技术”了。但即使新旧技术存在质的区别，其目标或目的却没有变化：实现相同的目标，或解决一样的问题。

然而这样教来学去的，抑或是有所发扬的，都只是技术而非技艺。技艺本身是与“人”相关的，它讨论的是人对技术的精通，而非技术本身。前者是量的问题，后者是质的界定。关于技艺的观点，应用到个体或群体都是合适的。例如杂技，学徒们是把套路当成技术来学来练的，如果有了一定的熟练度，便可以称之为技艺有成了。无论一个人的技艺与一个团队的技艺，都是讨论他们在某个技法/技术/套路上的熟练度的。

架构的确首先是一种实作的技术。这是毋庸置疑的，因为的确是在工程实践过程中产生了架构这一角色并承载了属于它的需求。这也是架构过程的“形成论与组成论”两个观点的真正出处。对于一个既存的架构，实作者认为它是源自于一个形成的过程，所以得到前一种观点，即架构的出处在于这些阶段的组合；而当实作者认为架构表达的是系统映像的具体内容时，便会得到后一种观点，即架构的落足在于这些内容的组成。

我们的确可以传授架构技术，并进一步地讨论架构的技艺问题。并且，无论是在架构活动中发生了质的还是量的变化，我们都可以归结于新技术的产生，或实践者的技艺日趋娴熟。然而，我们应该关注到这一活动的本质：无论是前者亦或后者，都是将架构作为一个死物，并试图通过模仿来复制一个新的架构。

一定程度上来说，这是有效的方法。但正如我对艺术的评价一样：艺术是不可能被“生产”出来的，生产出来的叫“艺术品”。通过复制的方法得到的架构失去了在形成论中的精髓，即映射与约束；也失去了组成论中的精髓，即关系与通信。即使我们通过某种过程将这些“精

髓”凝集在一个架构模式（以及由此而来的架构方法）之中，我们也失去了最原始的架构者的思想过程，例如我一直追寻的问题是：曹冲是如何想到了称象的方法？

二

即使 A 和 B 可以做同样的事，并产生同样的结果，我们对二者的认识也可能完全不同。例如，其中 A 可以是艺术家，A 的作品可以称为艺术品，A 的行为可以称为艺术；而 B 可能是机器，其结果是产品，其过程是生产。所以，如果仅以“过程与结果的相同”来考察，艺术家与机器就是同一的。

但这显然是笑话。因此，仅在技术与技艺的层面是无法定义“艺术”的——技术与技艺讨论的总是讨论结果的或过程的、质的或量的、部分的或全体的异同。

艺术一部分表现为独特性，但这种独特在本质里是表现为创作者的思想性的。艺术的思想性是很主观的，无法通过简单的表达来确证它。所以，一件因此而有趣的事是：梵高的手稿是艺术品，因为其中蕴含了——至少是我们认为它蕴含了——大师的思想；孩童的涂鸦也是艺术品，因为孩童在无意识间也加入了他的思想，即便这一思想单纯而又直白；但一个艺术系的学生的摹本就不是艺术品，如果这只是单纯地临摹而缺乏特有的理解与表达的话。

所以我常常说，即使你做出来的同样是一个三层（或 N 层）架构，如果你是通过系统分析、思考、权衡而得到这一架构决策的，那么它仍具有独特而丰富的架构思想；但如果只是因为与当前系统的背景类似，而使得你选择了这种架构形式，那么这只是一个工程师的技术选型，而非架构师的架构过程。

架构思想是对系统的认识的方法与结果(21)：从方法上来说，思想决定了如何认识系统；从结果上来说，思想表现为对系统的认识。若以艺术的眼光来看架构，必将以架构师在思想上的独特性为前提，进而得到他对系统认识的不同，以及对系统表达的不同。但即使是这样看过来，架构之与系统的形似，架构之于过程的有序，以及架构之于认识的深刻等，凡可度量规测的，都必落于形式之窠臼。

窠臼是思想的表达，形式是架构的道具。

三

文字在艺术面前是苍白的。

所幸我们创造了一个词汇来弥补这种苍白，这就是“美”。

关于技术、技艺与艺术，有三种美。其重点各有不同。首先，技术的美在于可行。一般来说，越是简单、越是易于理解的，其可行性也就越强。所以通常的，技术的美也称为“简单是美”，这种简单也包括有序与无序的重复。但总的来说，简单性可以体现在概念定义上的、抽象表达上的、结构组成上的以及可核查与重现上的等方面。这些方面必将最终表现为技术上的可行性与必然性(22)。

其次，技艺的美在于超越。技艺是基于技术的可行性而得来的，并通常通过精炼纯化的过程来得到。所以技艺的美也称为“极致是美”，这里的极致便是程度的用词。技艺必将围绕技术实施的过程来展开，因此技艺的美必然包括对过程性能与结果品质的提升（而非盲目的复制）。技艺与整个技术实施过程中的人——个体与群体——都相关，是他们于自身或于其他人的超越，因此精湛、纯熟、高超等关于技艺的赞美之词都是面向人的。技艺的美同样包括对“人+技术”的整体的、动态与静态的观感，例如，它既可以静止于一个人凝神专注的一霎，也可以记录于整个工作场所下的繁忙。所以，总的来说，技艺的美与过程相关，与过程中的人与技术相关，也与过程的结果相关。但对于这种种相关性中的“超越”，却是基于量性的变化与审度的。因为若是在质底里的变化，便已经超出了技艺表现的需求，而成为技术方法的需要了。

第三，艺术的美在于如一。艺术在底子里是艺术家的主观思想、理念，在表达上是艺术的形式，在过程上是技术与技艺，艺术的美在于这些方方面面的统一。艺术的独特性源起于艺术家对自我、外界以及“自我与外界的系统整体”的认识，艺术是因为这是认识的独特性而美的，是表达结果与原初认识如一而美的。若没有认识，则艺术是空洞的；若没有独特，则艺术是黯淡的；若没有如一，则艺术是不完美的。

若架构师确有思想，但无法表达出来或他的表达与思想并不一致，那么是不美的；若架构师拿出了一个“看似完美”的作品，却没有任何有意义的思想，那么也是不美的。若架构是一门艺术，则架构艺术的美，必以追求思想、形式、技艺完美如一为最终标准。

四

美的学问究其根底是讨论三类东西：美，美的对象，以及美的感受与意识(23)。

就“美的感受与意识”来说，感受是因客观对象影响而形成的主观认识，而意识则主要是主观反应(24)。我们对架构结果或过程的、所有可能的看法，都可以归为“美或不美”，即使是“正确性”，在一些人的眼中也可能加上“美或不美”这样的判定条件。但这样的感受只是肤浅的、皮表的。就架构结果来说，在架构图上加上一条线以使某种意思表达更确切，或者将一个模块分成两个或多个以使它便于实施，这些都可以使结果看起来更美一些，而无损于架构师的原意。架构师也可以“零代价地”变换这些表达手法，以满足不同的交流者的审美趣味。对此，我的意思是说，架构就感受与意识而言，如何使它“更美”，是可以去迎合沟通对象的，不必拘泥于“架构必须做成怎样”这样的前设。

因此，我常常会在白板上画下一个架构草图，这时我是不讲架构的材质的；会用 PPT 动画来模拟一个系统的演进，这时我是不讲架构的逻辑的；会直接交付一段代码来表达我在架构方面的设定，这时我是不讲架构的角色职能的；会长篇累牍地书写架构文档以满足某些官僚的要求，这时我是不讲实用性的，等等如此，架构在“感受与意识”的美既在于我之所见，也在于人之所见，这是在不能提高“环境关于架构的审美”的情况下的一时之选——但整体上，它应是无碍于架构师对架构的美的主观标准的。相反，若架构师在这些肤浅而皮表的问题上去纠缠“美与不美”的认识统一，“架构（这一过程整体）”便因丢掉了“大局观”而显得破败不堪了。

而这就渐渐地触及了“美的对象”这个话题。我们是否要求所有的部分都是美的，并且其整

体也是美的，并且部分之于整体也是美的……我们可以无限制地追求架构中的各种对象的美吗？仅仅对于艺术家来说，答案当然是“可以”。但正如我们在时装模特的身上看到的大多数“美的”服饰与妆扮，都不会出现在生活中一样，纯艺术论观念下的美也大多是不实用的——所以，事实上我也认为“架构艺术”对美的追求是显得有些“道化”的，而不是可行的、可作依据的、可于工程实施中去求索的。

从架构对于工程的意义、对于系统的意义以及对于一个实施团队的意义来说，无限制的、漫无目的地追求美是一种浪费。因此，我唯只将架构的美的对象定位于“时间与空间”两个维度。在时间维度上，我希望一个架构的美在于能以其持续性来保障系统的实施；在空间维度上，我希望一个架构的美在于能以其结构性来保障系统的成本。无论是软件产品还是硬件产品，对于这样一个系统，若既是可实施的又是成本可控的，或称为规模与复杂性可控的，那么该系统是否能最终完成便只需由必要性来决定了(25)。

就“‘美’是什么”这个问题来说，是一个哲学命题。哲学是“我见”(26)，其核心在于构建“我的”哲学认识体系。关于“美”这样一个具体的哲学命题，在此前的讨论中，我实际是将美的细节，例如“美的感受与意识”，摒弃了去。因为在我看来，若架构是系统所必需的映像，那么这架构也就必以反映“系统的系统性”为核心目的，以“系统的本质”为唯一正确的思考对象。从这一点上来讲，架构要做到的便是抹去那些枝节的东西，将系统主体的、正确的、无可争辩的事实揭示出来，因为唯有这些才能长期而又大范围地影响到系统的推进过程。而这样一来，架构在形式、感受与意识方面的美，便是次之又次的需求了。这些需求是可以并且也是需要通过后续的软件开发活动（例如设计）来补充的。需知设计所重的，正在于系统之细节刻画；而架构所重的，是先于刻画之前的、对系统之本实的确立。

当我们回到美的对象，亦即时间与空间下的架构，亦即探求其持续与结构上的美的问题时，我想尽我所能使用的词汇，尽我所能表达的认识，尽我所愿意接受的、对美的架构的最终审美标准来说，

“架构的美在于不朽”

应该是对此前讨论的所有架构原则的满足与契合，也是我对架构的所有认识的最终规约与展陈。若我在架构这一领域，对于架构师还有什么期冀的话，“做出不朽的架构”便是我最发自内心的赠言了。

五

通常，面对一个系统，一开始就讨论高并发、大流量、大数据以及大规模运算的架构师，是入门零段的。他还不懂得忽略与聚焦。

通常，面对一个系统的组成，大谈平衡与模型的架构师，是入门一段的。他还不懂得平衡只是技法，系统是没有平衡的，系统是在动态中不平衡地发展的；系统是一个时间轴上的东西，而非一个瞬间的衡态，例如模型。

通常，脱离了平衡的味趣，奔逐于系统的关键，寻求种种方案并努力实施的，是架构师的初段。这并没有不好，这些架构推进并演义了整个行业的瑰丽，如同那珠宝闪烁，成就了前台

的舞者。

通常，诠释着舞蹈之绝美的有两种人，一种是会审美的看客，一种是会创造美的编舞。他们都将自我之见作用于美的一片一片，如同架构师通过时间与空间的拼接来完成系统的全体。美与不美都任由评说，而又各有评说的标准。无论是作为看客还是编舞，这样的架构师已得架构之纲法精要。

通常，把舞蹈表现得完美无缺的，是一个舞者。那个舞者就是那段舞，当他表演的时候，编舞认为这段舞蹈是为舞者而生，而自己只是那个接生者；看客认为自己是舞者；舞者却从不承认这是表演。这样的架构师，他的架构对象和自己已成一体，但我很难找到一个人来诠释这一角色，因为他必已完美地谢幕。

其作品也必为不朽。

————————————————————————

(1) “形式化系统总是按如下顺序形成的：先确定有意义的符号，然后从符号中抽象掉意义，并用形式化方法构成系统，最后对这个所构成的系统作一种新的诠释。”这是著名哲学家与逻辑史学家波亨斯基（J.M.Bochenski）在《当代思维方法》中对形式化的方法的概括。

(2) 这里的意思是说，是否使用内存数据库，或者某种特定的数据结构（可以考虑结构化文件存储），或者特定的本地数据库等，这些方案的选择是由开发人员来决定的。

(3) 这里涉及两点，其一是所谓“特定……实现方案”，例如交付界面不能是“对于 a 用户，存取 personal.dat 文件以得到 user 数据结构中的 age 成员”，应当抽象为“对于用户名 a，存取 user 数据项（或 age 值）”；其二，这也意味着要求这样的开发人员有一定的系统设计能力。

(4) 这一观点事实上是对“自然语言是否是形式化的”的一个拷问。一定程度上来说，自然语言中“被形式化”的部分是我们对语言的形成及表达的阶段求解，而其他的则是我们尚显无知的部分。

(5) 事实上我已经想到了某种不以形式化为方式的表达，只是它并不适宜于我们这里讨论——不过即使如此，它与“架构基于概念抽象”仍然是不悖的。

(6) 我的意思是说，即使是同一个开发人员，他也必然面临“现在的系统”与“以前的系统”之间的差异。这种差异的表现手法之一是版本化的源代码，之二则是不同阶段的“架构”。就其实效性来说，后者在讨论“整体差异”方面是更有优势的。

(7) 如果是一个人自言自语，是我之我的交流；如果是自我的反省，是我之于故我的交流；如果是我的畅想，是我之于势我的交流（此势者，至而未至也）。总而言之，交流总是有主客体的，无论是彼此之别，或一己之侧相。

(8) 这并非不存在，例如在架构师团队中基于“体系架构”来讨论部署就是常见的事情。问题在于这一讨论中的信息——抽象对象及其概念——过于粗略，难于实际地指导部署过程，因此也就不能取代“部署架构”来与部署人员交流。

(9) 因此我并不能确切地说“世界上所有称为架构的东西”都适于本书的讨论（尤其是指总论与上一章的讨论）。

(10) 在实践中，我通常会要求架构师试图站在两个点上考虑系统脉络，一是核心数据的流向，二是用户行为的起始。这许多时候决定了整个系统中的、相对长期不变的东西。

(11) 在实践中，我通常会要求架构师以“一句话或一个标题”来定义他的系统。我们最终必须关注这句话或这个标题对于系统的概括力与约束性，而非去感觉它是否醒目或时髦——后者通常是产品经理的事情，并且常常为市场经理以及大老板所乐见。

(12) 微之甚微，巨之愈巨，皆是系统规模的增加。

(13) 注意，这里并没有用“生产过程”。在一定程序上来说，“生产”是有特定的工程含义的。

(14) 引自《丰田汽车案例：精益制造的 14 项管理原则》，杰弗瑞 • 莱克著。

(15) 然而我们现在的过程论者，常常会忽略了生产型企业中的“研制”过程，以及研制的本质是“通过大量的错误修正来得到正确的步骤”这一事实。

(16) 可以理解为生产过程中，基于某个产品原型的具体生产工序。

(17) 有两个原因，其一是目前的模型表达法没有参数化，其二是工序本身不是组成论视角下的结论。

(18) 我们此前的绝大多数讨论都是基于这两个设定的。

(19) 主体价值是形成系统的核心驱动力量，持续性、复杂性或可变性只是这一过程中的种种表现而已。

(20) 在这样的系统中，是无法且不必讨论“佛陀拈花”这一问题的。

(21) 这在前面所谈到的五条架构原则中，是有相当充分的体现的。

(22) 这里的必然性是指：即使当前是不可行的，在特定的技术条件下也必然是可行的。

(23) 引自《美学概论》，陈望道著。

(24) 感受与意识的不同，大抵在于前者源起于外，后者发端于内。从认识论的角度上来说，

这是“知己觉”和“觉未觉”的区别。

(25) 架构只说明可能性而并不说明必要性，后者是一个产品/业务/企业决策的问题。

(26) 因此任何一个人谈及哲学，都是他之于哲学的认识。这一认识不必有强加性，也不必求同。

附录一

做人、做事，做架构师——架构师能力模型解析(1)

架构是一个从全局到局部的过程，而实施正好反过来，是从局部到全局。这也正是“设计做大，实施做小”的另一个层面的含义。“设计大”才可以见到全局，才知道此全局对彼全局的影响；“实施小”才可能关注细节，才谈得上品质与控制。

事实上，大多数情况下架构是在为“当前项目之外”去考虑，这可以看成全局关注的一个组成部分。因此我们需要界定所谓“全局”的范围：若超出公司或整个产品系列、产品线或产品规划的范围，则是多余的。所以当架构决策谈及“全局”时，其目标并不见得是“保障当前项目”，而又必须由当前项目去完成。(2)

一个经常被用到的例子是：如果仅为当前项目考虑，那么只需要做成 DLL 模块；如果为产品线考虑，可能会是“管道＋插件”的结构形式。而“管道＋插件”的形式显然比做成 DLL 模块更费时，这个时间成本（以及其他成本）就变成了当前项目的无谓开销。

这种全局策略对局部计划的影响是大多数公司不能忍受的，也被很多团队所垢病。然而这却是架构师角色对体系的“近乎必然”的影响——如果你试图在体系中引入架构师这个角色的话。一些情况下，体系能够容纳这种影响，例如“技术架构师”试图推动某种插件框架，而正好开发人员对这项技术感兴趣，那就顺其自然地花点工夫去实现了。但如果实施者或实施团队看不到“多余的部分”对他们的价值，来自局部的抵触就产生了。

这种情况下，平衡这些抵触就成了架构推行的实务之一。在我看来，“平衡”是全局的艺术和局部的技术。也就是说，一方面架构师要学会游说，另一方面也要寻求更为简洁的、成本更小的实现技术。只有当整个体系都意识到（你所推行的）架构的重要性，而且实施成本在他们可以接受的范围之内时，他们才会积极行动起来。

所以所谓平衡，其实也是折中的过程。构架师只有眼中见大，才知道哪些折中可以做，而哪些不能。所谓设计评估（模型图(3)中的实现能力→设计能力→设计评估分支）并不是去分析一个设计结果好或不好，而是从中看到原始的需求，看到体系全局的意图，然后知道在将

设计变得更为“适当”时可以做哪些折中。同样的原因，架构师也必须知道自己的决策会产生的影响，才能控制它们，以防它们变成团队的灾难。有些时候，架构师甚至需要抛弃一些特性，以使得项目能够持续下去，因为产品的阶段性产出只是整个战略中的一个环节，而不是全部。

————————————————————————

(1) 节选自《程序员》杂志 2008 年第 4 期同名文章。

(2) 通常这是指系统架构的一部分工作，但一些技术架构可能也需要这样的全局眼光才能正确决策。

(3) 这里的模型图是指与本篇文章同时发表的一个“架构师能力模型”，并非本书中所论的模型。该模型可参阅《程序员》杂志中的原文或我的个人博客。

附录二

专访：谈企业软件架构设计（节选）(1)

企业实施过程中的架构问题，可以分成两个部分来考虑：一个是软件企业自身，另一个是工程的目标客户（有些时候它与前者一致）。基本上来说，架构设计首先是面向客户的，甚至在整个工程的绝大多数时候都面向客户。因为理解决定设计，所以让架构师尽可能早地、深入地了解工程目标、应用环境、战略决策和发展方向，是至关重要的；否则，架构师是不可能做出有效的设计来的。

架构设计关注于三个方面：稳定、持续和代价。

稳定性由架构师的设计能力决定。架构的好坏是很难评判的，但基本的法则是“适用”。如果一个架构不适用，那么再小或者再大都不可能稳定。因此进一步推论是“架构必须以工程的主体目标为设计对象”。看起来这是个简单的事，但事实上很多架构设计只是在做边角功夫，例如为一两处所谓的“精彩的局部”而叫好，全然不顾架构是否为相应的目标而做。

持续性由架构师的地位决定。如果不能认识“设计的一致性”，以及架构师对这种一致性的权威，那么再好的架构也会面临解体，再长远的架构也会在短期内被废弃。架构的实施是要以牺牲自由性为代价的，架构师没有足够的地位（或权威）则不可能对抗实施者对自由的渴望。通常的失败并不在于架构的好或坏，而是架构被架空了，形同虚设。

代价的问题上面有过一点讨论，但方向不同。这里说明的是，如果架构师没有充分的经验，不能准确评估所设计的架构的资源消耗，那么可能在项目初期便存在设计失误；也可能在项目中困于枝节，或疏离关键，从而徒耗了资源。这些都是架构师应该预见、预估的。

对于企业设计来说，上面三个方面没有得到重视的结果就是：迟迟无法上线的工程、半拉子工程和不停追加投资的工程项目。我不否认项目经理对这些问题的影响，但事实上可能从设计就开始出了问题，而项目经理只是回天乏术罢了。

最后说明一下，我认为目前大多数的企业项目都缺乏架构上的考量。大多数软件公司只是出于自身的需要（如组件化和规模开发）而进行架构设计。这样的设计不是面向客户的，事实上这增加了客户投资，而未能给客户项目产生价值。这也是我强调架构面向客户的原因之一。

————————————————————————

(1) 节选自 ZDNET 网站 2007 年 3 月对本书作者的专访。

参考资料

专业著作与文献

《结构程序设计》，陈火旺、吴明霞、丁宪理译，科学出版社出版，1980 年。

O.-J.Dahl，E. W Dijkstra and C. A. R Hoare, Structured Programming，Academic Press Ltd.，1972

《程序设计语言：实践之路》，裘宗燕译，电子工业出版社出版，2005 年（第 1 版），2007 年（第 2 版）。

Michael L. Scott, Programming Language Pragmatics，Second Edition，Morgan Kaufmann，2005

《符号理论的基础》，周礼全译。原译名为《指号理论的基础》，收入《资产阶级哲学资料选辑 第 18 辑》，由《哲学研究》编辑部编，上海人民出版社出版，1966 年。后收入《莫里斯

文选》，涂纪亮编，社会科学文献出版社出版，2009 年。

Charless W. Morris，Foundations of Theory of Signs（1938），Chicago University Press，Chicago，1955

《计算机编程的概念、技术和模型》，无中文版。

Peter Van Roy and Seif Haridi，Concepts，Techniques，and Models of Computer Programming，The MIT Press，2004

《认知（第二版）》，无中文版。

Arnodl Lewis Glass and Keith James Holyoak，Cognition. 2nd Edition，Random House，1986

《编码：隐匿在计算机软硬件背后的语言》，左飞、薛佟佟译，电子工业出版社出版，2010 年。原译名为《编码的奥秘》伍卫国、王宣政、孙燕妮等译，机械工业出版社出版，2000 年。

Charles Petzold，Code: The Hidden Language of Computer Hardware and Software，Microsoft Press Redmond，WA，USA，2000

《美学概论》（1927）是我国最早探究形式美的美学论著之一，陈望道著。收于《陈望道学术著作五种》，复旦大学出版社出版，2005 年。

《数据结构》，严蔚敏、吴伟民编著，清华大学出版社出版，1987 年。

《人月神话（32 周年中文纪念版）》，UML China 翻译组　汪颖译，清华大学出版社出版，2007 年。

Frederick P. Brooks，Jr.，The Mythical Man-Month，Essays on Software Engineering，1975

《人件（第 2 版）》UML China 翻译组译，清华大学出版社出版，2011 年。

Tom DeMarco and Timothy Lister，Peopleware: Productive Projects and Teams（Second Edition），Dorset House，1999

《当代思维方法》，童世骏、邵春林、李福安译，上海人民出版社出版，1987 年。

J. M. Bochenski, The Methods of Contemporary Thought，1965

《丰田汽车案例：精益制造的 14 项管理原则》，李芳龄译，中国财政经济出版社出版，2004 年；再版名为《丰田模式：精益制造的 14 项管理原则》，李芳龄译，机械工业出版社出版，2011 年。

Jeffrey K. Liker, The Toyota Way，McGraw-Hill，2003

《软件工程——实践者的研究方法（第四版）》，黄柏素、梅宏译，机械工业出版社出版，1999 年。该书第六版由郑人杰、马素霞、白晓颖等译，机械工业出版社出版，2007 年。

Roger S. Pressman, Software Engineering ， A Practitioner's Approach ， Fourth Edition, McGraw-Hill，1997

《软件工程（第八版）》，程成、陈霞译，机械工业出版社出版，2007 年。

Ian Sommerville, Software Engineering，Eighth Edition，Addison-Wesley，2006

《你的灯亮着吗？——发现问题的真正所在》，章柏幸、刘敏译，清华大学出版社出版，2004 年。

Donald C. Gause and Gerald M. Weinberg, Are Your Lights On? How to Figure Out What the Problem Really Is，Dorset House，1990

《成为技术领导者——解决问题的有机方法》，朱于军、李先华等译，清华大学出版社出版，2003 年。

Gerald M. Weinberg， Becoming A Technical Leader： An Organic Problem-Solving Approach，Dorset House，1998

《动机与人格》，许金声等译，初版由华夏出版社出版，1987 年。第三版由中国人民大学出版社出版，2007 年。著名的马斯洛人类需求五层次理论（Hierarchy of Needs）提出于他在 1943 年发表的论文《人类动机的理论（A Theory of Human Motivation）》，并在《动机与人格》一书中进行了全面论述。

Abraham Harold Maslow， Robert PH. D. Frager and James Fadiman, Motivation and Personality（1954），HarperCollins Publishers; 3 Sub edition（January 1987）

《观止——微软创建 NT 和未来的夺命狂奔》，张银奎译，机械工业出版社出版，2009 年。

G. Pascal Zachary, Show Stopper!: The Breakneck Race to Create Windows NT and the Next Generation at Microsoft，Free Press，1994

《梦断代码》，韩磊译，电子工业出版社出版，2008 年，再版于 2011 年。本书中引用的页码为 2008 年版。

Scott Rosenberg，Dreaming in Code: Two Dozen Programmers，Three Years，4732Bugs，and One Quest for Transcendent Software，Three Rivers Press，2008

《集装箱改变世界》，姜文波等译。由机械工业出版社出版，2008 年。

Marc Levinson, The Box，Princeton University Press，2006

《程序员修炼之道——从小工到专家（评注版）》，周爱民、蔡学镛评注，电子工业出版社出版，2011 年。

Andrew Hunt and Dave Thomas, The Pragmatic Programmer: From Journeyman to Master，1999

《大道至简》，周爱民著，电子工业出版社出版，2007 年。本书有关“具体工程”的根据提出于该书第三版（点评版），电子工业出版社出版，2010 年。

参考文献论著

《逻辑的数学分析——关于演译推理的一篇随笔》，乔治·布尔，1847 年。

George Boole，“The Mathematical Analysis of Logic， Being an Essay Towards a Calculus of Deductive Reasoning”，1847

《逻辑的演算》，乔治·布尔，1848 年。

George Boole，“The Calculus of Logic”，1848

《继电器和开关电路的符号分析》，克劳德·艾尔伍德·香农，1938 年。

Claude Elwood Shannon，“A Symbolic Analysis of Relay and Switching Circuits”，1938

《通信的数学原理》，克劳德·艾尔伍德·香农，1948 年。

Claude Elwood Shannon，“A Mathematical Theory of Communication”，1948

《管理大型软件系统开发》，温斯顿·罗伊斯，1970 年。

Winston W. Royce，“Managing the Development of Large Software Systems”，1970

《软件架构的“4+1”视图模型》，菲利普·克鲁奇顿，1995 年。

Philippe Kruchten，“Architectural Blueprints - The ‘4+1’ View Model of Software Architecture”，1995

《论将系统分解为模块的准则》、《设计易于扩展和收缩的软件》与《复杂系统的模块化架构》，Parnas 关于信息隐蔽理论的主要文献，无中文版。

David Parnas，“On the Criteria to Be Used in Decomposing Systems into Modules”，“Designing Software for Ease of Extension and Contraction” and “The Modular Structure of Complex Systems”

《社会传播的结构与功能》，洪允息、黄林译（节选），收入《传播学（简介）》，人民日报出版社出版，1983 年。其后，在《西方新闻传播学名著选译》一书中，顾孝华选译了 1-5 与 7-9 节，上海社会科学院出版社出版，2008 年。

Harold D. Lasswell，“The structure and function of communication in society”，1948

《软件构架实践》，孙学涛、杜学绘、刘冬萍译，清华大学出版社出版，2003 年

Len Bass，Paul Clements and Rick Kazman, Software Architecture in Practice，Addison-Wesley，1997

《软件构架评估》，孙学涛、朱卫东、赵凯译，清华大学出版社出版，2002 年

Paul Clements，Rick Kazman and Mark H. Klein, Evaluating Software Architectures: Methods and Case Studies，Addison-Wesley，2001

《Java Web Services 架构》，无中文版

James McGovern，Sameer Tyagi，Michael Stevens and Sunil Mathew，Java Web Services Architecture，Morgan Kaufmann，2003

《程序开发心理学（银年纪念版）》，邓俊辉译，清华大学出版社出版，2003 年。

Gerald M. Weinberg, The Psychology of Computer Programming: Silver Anniversary Edition，Dorset House，1998

《组织行为学：工作中的人类行为（第十版）》，陈兴珠、罗继等译，经济科学出版社出版

John W.Newstrom，Organizational behavior，Human Behavior at Work，McGraw-Hill/Irwin; 13 edition，2010

《社会契约论》，（法）卢梭（Jean-Jacques Rousseau）著，何兆武译，汉译世界学术名著（丛书）本，商务印书馆出版，2003 年。

《权力与繁荣》，（美）奥尔森著，苏长和、嵇飞译；上海人民出版社，2005 年。

Mancur Olson，Power And Prosperity: Outgrowing Communist And Capitalist Dictatorships，Basic Books，2000

《集体行动的逻辑》，（美）奥尔森著，陈郁、郭宇峰、李崇新译，上海三联书店、上海人民出版社出版，2011 年。

Mancur Olson，The Logic of Collective Action: Public Goods and the Theory of Groups，Harvard University Press，1971

历史与革命文献

《中国史学名著》，钱穆著，三联书店出版，2000 年。

《史记》，《全校全注全译全评史记》（六卷本），杨钟贤、郝志达主编，天津古籍出版社出版，1997 年。

《韩非子·三人成虎》，韩非著。

《三国志·曹冲称象》，陈寿著。

《吕氏春秋·刻舟求剑》，吕不韦主持编撰。

《毛泽东文集》（八卷本），中共中央文献研究室编，人民出版社出版，1996 年。

《朱德选集》，中共中央文献研究室编辑委员会编辑，人民出版社出版，1983 年。

《历史选择了毛泽东》，叶永烈著，广西人民出版社出版，2005 年。

图表索引

附图索引

总论：领域角色的关注

图 0-1　EHM 图及其隐含着的轴向

图 0-2　将 EHM 图抽象为单向的轴线

图 0-3　轴线中的经营角色

图 0-4　模型 1：对 EHM 模型进一步抽象所得的新模型

图 0-5　模型 2：“工程的组织视角”对模型 1 的影响

图 0-6　模型 3：目标在模型 2 中的投影关系

图 0-7　模型 4：对模型 3 的概念修正

图 0-8　模型 5：组织视角下的工程视图

图 0-9　模型 6：不同规模的系统在 VEO 模型上的映像

图 0-10　软件工程的质量三要素和体系层次

图 0-11　模型 7：系统局部与全局的关系

图 0-12　技术、管理之于架构是一体两面的关系

图 0-13　基于实施视角，对传统工程模型的另一种抽象描述

图 0-14　模型 8：组织结构调整带来的问题

图 0-15　模型 9：积极的组织适应带来的效果

图 0-16　模型 10：消极的组织对抗带来的平衡态势

图 0-17　模型 11：架构角色的职责可能对组织产生的影响

图 0-18　模型 12：面向合作的组织适应带来的效果

图 0-19　（与领域相关的）领导力下的衡态

图 0-20　外层角色的影响

图 0-21　更多的组织角色构成的阶层

图 0-22　通过合并来打破阶层（示例 1）：决策者的思想

图 0-23　通过合并来打破阶层（示例 2）：实现者的思想

图 0-24　通过合并来打破阶层（示例 3）：尝试者的思想

图 0-25　通过合并来打破阶层（示例 4）：规划者的思想

图 0-26　组织力与领导力对组织形态的不同影响

第二篇：程序源流：从计算到系统（上）

图 2-1　三种逻辑表示

图 2-2　开发规模的不同等级

图 2-3　“应对开发规模”的结构化求解：四种等级的本质

图 2-4　单元、库与界面间的关系

图 2-5　面临的问题与背景：Application 在“结构化”上选择不同方向的原因

图 2-6　两种“结构化”方向的本质差异

图 2-7　已知数据内容的顺序存储

图 2-8　数据内容未知时，预占“存储地址值”所需的确定空间

图 2-9　保证顺序存储的方案：通过填写预占位置，指示实际的存储位置

图 2-10　一个简单的哈希算法在存储上的映射

图 2-11　指针在哈希表上的运用

图 2-12　哈希表的结构与概念间的抽象关系

图 2-13　验证：保留 Name/Key 的原因

图 2-14　从一个入口开始：操作系统准备的执行环境

图 2-15　语法树：对语法元素进行的排序

图 2-16　生成的语法树（为分析“基于数据的排序”而进行了排版变化）

图 2-17　基于数据的排序：将“数据”理解为运算所需的参考

图 2-18　“结构”与“对象”在抽象本质上的一致性

图 2-19　PME 作为语言特性所实现的抽象概念

图 2-20　基本的对象继承

图 2-21　对象继承性的衍化与证明

图 2-22　X 与 C 是范围{B}中同类对象的多态

图 2-23　继承性决定了 X1、X2 与 C 在范围{B}中的多态关系

图 2-24　对象的数据之于行为的可见性问题

图 2-25　对象之于系统的可见性问题

图 2-26　对象在继承层次上的可见性问题

图 2-27　跨系统继承（对象复用）时的可见性问题

图 2-28　GOF 模式的基本分类

图 2-29　模式是对视察所见关系的一种抽象描述

图 2-30　GOF 隐含地基于并依赖“类继承”模型

第三篇：程序源流：从计算到系统（下）

图 3-1　应用开发中三个层次上的需求

图 3-2　从“代码的粒度”出发的抽象概念

图 3-3　基本功能：将 m 变换为 n

图 3-4　分解：三个步骤

图 3-5　持续分解：更多的步骤、逻辑或子系统

图 3-6　“类”的价值与局限：对传统组织方式的一种替代

图 3-7　（产品的）交付形式相关的组织方式

图 3-8　“建模者”所面临的主要需求

图 3-9　敏捷工程并不能消灭上述需求

图 3-10　项目要素之间的平衡三角

图 3-11　管理平衡三角

图 3-12　MSF 的三种模型与项目要素之间的关系

图 3-13　面对两种需求背景，所提出的主要（实施）概念

图 3-14　实施层面的思想、技术与具体方法

图 3-15　将状态从纯数据中剥离，并讨论与逻辑步骤（Step）间的关系

图 3-16　消息的发布与处理：在整个逻辑步骤（Step）链条上的数据隔离

图 3-17　数据在计算机语言中的种种抽象与概念递进关系

图 3-18　约定规格：值性质下（状态与消息）与变量性质下（结构与对象）的抽象概念对比

图 3-19　两种“PD 模型”：数据或逻辑的不变性

图 3-20　两种可选可替代的并发模型：共享状态模型与消息传递模型

图 3-21　对确定数据下的 PD 模型的两种不同理解

图 3-22　三种登录模型：单数据库方案（左）、增加应用服务器的方案（中）、增加数据镜像的方案（右）

图 3-23　从“集中式”向“分割式”过渡：分表分库的方案

图 3-24　“数据结点”作为独立系统层次中的可部署对象

图 3-25　传统的三层或多层系统架构

图 3-26　图 3-25 中架构的“P-D 关系”模型

图 3-27　分布式框架：对界面（对规则的封装）的思考

图 3-28　分布式框架：领域问题及其方案带来的思考

第四篇：架构的思想与指导原则

图 4-1　对架构师可能关注的话题的分类

图 4-2　通用办公系统架构 v0.0…1

图 4-3　通用办公系统架构 v0.0…2

图 4-4　通用办公系统架构 v0.0.0.1

图 4-5　认知过程的方法树

图 4-6　认知过程的方法树：建立知识的两种方法

图 4-7　通用办公系统架构 v0.0.0.2

图 4-8　基于对思考行为（的模式）的观察：上述概念或观点的层进关系

图 4-9　“程序过程式”的思维活动

图 4-10　面向问题的思维活动

图 4-11　架构师能力的三个方面

图 4-12　个人能力取向与职业倾向的对比

图 4-13　领域背景知识与可应对的架构规模的关系

图 4-14　架构师能力是对三方面能力的平衡性的要求

图 4-15　在“架构整体”上所需要的决策过程

图 4-16　多个实施阶段下的架构决策过程

图 4-17　一个可参考的架构形成模型 M0

图 4-18　通用办公系统架构 v0.0.0.2

图 4-19　一个概要的系统架构的模型

图 4-20　服务层的模型

图 4-21　框架层的模型

图 4-22　产品层的模型（面向产品的集成架构）

图 4-23　对“架构形成模型 M0”的简化：系统架构的一般过程

图 4-24　一个概要的系统架构的模型

图 4-25　整体划分成两部分

图 4-26　Spring 架构图

图 4-27　Spring 架构图：ORM 与 JEE 之间的领域与分类依据

图 4-28　Spring 架构图：AOP 与 Core 之间的层次与分类依据

图 4-29　Spring 架构图：JEE 的两种图示

图 4-30　在平面上表达领域的三种方法

图 4-31　并列结构的表现形式

图 4-32　嵌套结构的表现形式

图 4-33　以 B_C 之间存在关系为例改写架构的表达

图 4-34　嵌套结构与并列结构

图 4-35　其他结构在层次结构中的含义

图 4-36　并列结构之间的依赖

图 4-37　将领域转置为层次带来的依赖关系

图 4-38　层间依赖：对数据时序依赖关系的分析与解构

图 4-39　层间依赖：对逻辑时序依赖关系的分析与解构

图 4-40　层间依赖：归并

图 4-41　层间依赖：对嵌套结构转化为层次结构的处理

附表索引

第一篇：具体工程下的组织与行为

表 1-1　对三种软件产品定义及其需求类型的简单考察

第二篇：程序源流：从计算到系统（上）

表 2-1　PC 中的设备与其理解的数据

表 2-2　服务：比较生活中的邮寄与软件开发中的会话

表 2-3　服务：后台传输服务在 Windows 中的提供模式

表 2-4　服务：与上述服务等价的服务在网络环境下的提供模式

表 2-5　不同视角对四种开发规模的认识

表 2-6　语法元素的性质

表 2-7　GOF 模式：对结构型的分析

表 2-8　GOF 模式：对行为型的分析

第三篇：程序源流：从计算到系统（下）

表 3-1　模型要素在理论框架与集成环境中的关系

第四篇：架构的思想与指导原则

表 4-1　架构师在思维过程中使用的工具

表 4-2　面向架构话题的考察：授权与角色规则化带来的主要收益

表 4-3　架构规划在不同阶段中的重点

表 4-4　架构意图在不同阶段中的变化与入手点

表 4-5　三类界面原则的示例：Erlang 的一些实践性原则