图灵电子书

我的架构思想

基本模型、理论与原则

周爱民 著

内容提要

本书以系统的认识论作为出发点，全面描述了架构的思想、过程、方法。在此基础上，提出并论述了普遍性的架构理论和参考原则。通过全新的观察视角，本书对“架构”以及“架构师角色”提出了新的定义，并主要讨论了形成论与组成论两种架构方法。此外，本书在“架构意图”方面的讨论，既直指架构思想的本质，又为架构决策与实施找到了依据，颇为难得。

最后，本书在附录中提供了一个超越软件架构的案例，尝试解决作者早期提出的 EHM 模型（软件工程中的层状模型）中所蕴含的问题，由此提出了新的组织架构。

序 1：周爱民的道到底有多大多易

【一】

您现在读到的这本书，出自初版于 2012 年的《大道至易》。后者还有个前传，叫《大道至简》，作者自然也是周爱民，写于 2003 到 2004 年期间。那个时候，“架构师”这个角色在国内 IT 圈还不是特别为人所了解。而当《大道至易》问世的时候，架构师红极一时、满大街都是架构师的盛况已经开始走下坡路了，产品经理取而代之成为行业宠儿，而紧随其后的是那一轮移动互联网创业狂飙，再往后是数据科学家、人工智能专家集齐万千宠爱的 AI 热。长江后浪推前浪，一浪更比一浪浪，架构师这个职业，正如其他“先前浪”的职业一样，早就卧倒在沙滩上。只有周爱民这样的人，还在傻不拉几的搞他的架构师“大道”，从沙土中昂起头颅辩白说，我们架构师没有趴下，只是在做俯卧撑。不单如此，他还逼着我看他的辩词，还要写读后感。

我多年前确实花了一点时间了解架构模式，但离开技术社群日久，

早已生疏和淡忘，Doug Schmidt 的多卷本 POSA 是我对于软件架构模式的最后印象。坦率地说，阅读这本不大但也不易的作品，超过我目前对于软件架构的理解水平。所幸我近来一直关注系统方法论及相关的认识论话题，于这本书通读之后，倒觉得有些关联。而周爱民本人又是一个哲味十足的技术人，因此对于我从这个高空且虚泛的角度来谈他的这本书，他倒也能接受。

首先要指出，这本书并不好懂。所谓大道至易，这个易字，要让我来说，并非“容易的易（easy）”，毋宁理解为“变化之易（change）”。大道至易，就是大道的变化至多。变化多就难把握嘛，怎么会容易？中国古代文字，将 easy 与 change 用同一个易字表达，其中玄机我难以参透。不过我于本书中感到作者的自信，若是你能理解这本书中的大道，或许架构设计相对来说就会简单一些，而读懂这本书就变成了主要的挑战。

这本书确实较难读，作者所用的很多词汇，与我们生活中甚至一般专业技术讨论中使用的词汇形同而意异。此现象在本书的前半部分尤其突出，所以读书中的文字时，要格外认真，必须搞清楚一个词、一句话的确切所指。作者虽然在很多地方也强调了这个问题，但是这仍然使读此书变得颇具挑战。这也就意味着，读这本书需要耗费较大的心力，绝非枕边消遣之读物。

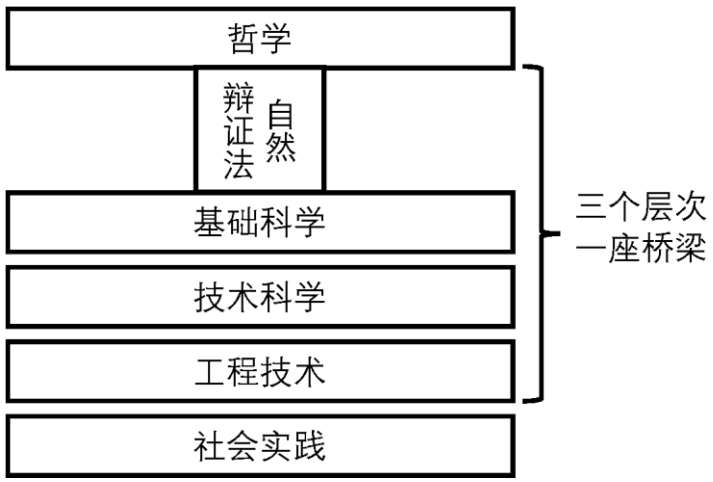
【二】

那么为什么这本书值得耗费心力来读懂呢？因为这是一本架构思维的认知升级之作。为了说明这一点，请容我用一点篇幅来谈谈认知升级这件事。

何谓认知升级？近来坊间围绕这个概念有一些讨论，不过主要是在

成功学的意义上进行的，不够严谨。事实上，关于认知的层次，学术界也有过相对严谨的研究。比如心理学大师布鲁姆将人类认知按照复杂程度排列为记忆、理解、应用、分析、评价、创造六个层次，而钱学森在上世纪八十年代将现代科学技术总体系放在一个统一框架下，提出著名的“三个层次一座桥”的自然科学体系一般框架，也就是将科学技术分解成工程技术、技术科学和基础科学三个层次，并通过自然辩证法这座桥梁对接辩证唯物主义的哲学。可以表达为下图：

> （序）图 1 钱学森提出的学科体系一般框架



我们将认知升级放在这个框架里来讨论，显然比用“知之为知之，不知为不知，是知也”这样的诸子箴言的方式来讨论，更加清晰、更加精确。不过，今天互联网人群所面对的主要问题，无论是本书关注的软件和业务系统架构，还是企业经营、产品创新、市场战略、产业生态构建等等，其实都并非纯粹的自然科学和工程技术问题，而是要求我们在复杂的系统之内进行综合性的思考和实践，其中既包含经验的部分，又包含科学技术的部分，还包含哲学思辨的部分。为此，我们谈论认知升级，需要一个不同的框架。我略作归纳，将

这个新框架中的认知层次以 L0 至 L3 表示，其意义如下表所述：

> （序）表 1 本序文中提出的认知的四个层次

	名称	描述	举 例
L3	认识论	从哲学高度上分析我们在所面对的领域之内是否能够以及为什么可以产生科学的认知，我们认知的边界在哪里	黑箱认识论
L2	方法论	能够指导我们产生科学方法的方法和思维模式	系统论和系统科学
L1	科学方法	一套具体的、一致的、系统的概念、方法和过程，重复这个过程可以产生局部或全局最优的结果	各领域内经过实践检验的框架、方法、流程、规则
L0	实践经验	未经审视和加工的实践经验，包含真知灼见，但也包含大量的矛盾、混乱、随意和武断，各种成分纠缠在一起，往往将有效的实践变成一种不可重复的艺术行为	各领域的实践经验、管理艺术、领导艺术、成功学、心灵鸡汤、诸子格言等

【三】

然而，认知升级对我们具体有什么意义呢？在这里我用时下火热的机器学习认知升级过程来介绍一下。

机器学习的兴起，最直接的原因是大数据的出现。在大数据出现之前，大量的业务决策过程是所谓“拍脑袋”式的，也就是依靠经验和直觉。虽然数据很早就进入了商业管理的领域，但是时至今日，即使在全球领先的五百强企业当中，很多关键的决策，也只是把数据统计出来，变成图表，然后由“有经验”的管理人员根据数据来“拍脑袋”。根据上面的框架，这种认知水平，无论如何也只是停留在 L0 层次上。

相比之下，当前正在兴起的机器学习方法，无疑是一个巨大的进步。当前的机器学习工程师，未必需要深入理解机器学习的数学原理，只要根据一套完善的“套路”，配合类似 R 语言或者 Python 的 scikit-learn、TensorFlow 等工具，就可以开发出一个回归或者分类模型，帮助商业人士决策。这个就属于 L1 级的认知，对应到钱学森“三个层次一座桥梁”框架当中的工程技术层次。

而 L1 认知层次的局限性在于，一旦这个“套路”失效，只停留在这个认知层次的工程师，会束手无策，因为他们并不知道自己使用的方法和套路是怎么来的，自然也无法变通和调试。因此一般来说，我会建议学习机器学习的人，要有雄心达到 L2 层次，也就是通过理解“套路”和模型背后的原理，特别是数学原理，进入到方法论的层次。到了这个层次，你才能够在实践中知其然且知其所以然，恢恢乎游刃有余，才能称得上是“高手”，才算是一名数据科学家。

就解决具体问题来说，认知达到 L2 层次的数据科学家已经非常强悍。如果说他们的局限性，就在于他们对于自己的这一套做法，还缺乏“反躬自省”的审视。机器学习为何竟然是有效的？其有效性的边界在哪里？对什么样的问题可能会失效？这样的问题当然不需要每一个机器学习专家都去考虑，但是在整个机器学习社区，一定要存在一些思考这些问题的“哲学家”，他们能站在上帝视角来审视自己的学科本身，考虑这门学问的根本问题和长远命运。这种人就站到了 L3 层次，也有机会成为整个学科的领军人物。

【四】

当然，本书的中心话题是架构，并不是如上所讨论的数据科学。不过触类旁通，在架构的认知上，同样存在从 L0 向 L3 升级的问题。

在“内容简介”当中，本书开宗明义地说“本书以系统的认识论作为出发点”，正是提示读者，这是一本触及了 L3 级认知层次的书。虽然书中并没有明确的引用系统论、控制论、信息论等经典理论，但是从内容来看，作者所强调的正是系统论的认知思维模式。例如，系统论和控制论所基于的认识论是黑箱认识论，也就是通过客体输入与输出的关系，推测客体的内部的结构和联系，并且将这种推测假说表达为模型。而本书作者在第 1.1 节中描述的那个光线射进黑屋子的场景，对于“知得”与“识得”的辨析，对于“建立知识”过程的思考，以及对于这一过程在架构设计中的不足之处的论断，正是在 L3 层次去讨论架构。我们且不论书中的观点正确与否，站在这个认知层次上谈论业务和软件系统架构的书，这本书就算不是唯一，也肯定是稀罕的。硬币的另一面是对读者的要求。如果读者不能站在这个高度去读这本书，肯定也难以理解作者用词的考究。

当然本书并非仅仅 L3 层次上的作品。作者花了这本书的主要篇幅来介绍方法论，也就是 L2 层次的内容。毕竟大多数架构师无意成为架构师中的哲学家，而是希望提升自己解决问题的能力。这本书的第 5 章到第 7 章，构成方法论内容的主体，自然也是作者多年思考的萃集，值得读者花精力去琢磨。

比如说，在第七章列举的第三原则中，将架构表达为范围和连接件之合集。这里的妙处在于“范围”这一概念的把握。一般来说，我们将系统表述为子系统以及子系统之间的连接，但什么是子系统？在一个完整的大系统中，你凭什么将这一部分拎出来单独命名为一个子系统？系统论指出，系统以及子系统并不是一个客观的实体，而是人为规定的。在系统的边界之内的各部分，耦合关系比较紧密，而系统与其边界之外的部分，耦合比较松散。我们再思考一下这个说法，这不正是作者所说的“范围”吗？

同样的，第四原则说“过程之于结果，并没有必然性”，这也是控制论中讨论的主题之一。控制论告诉我们，一个控制过程能够达到预期的目标，是有条件的，例如需要有充足的信息，需要高效的负反馈结构，等等。细读本书在这一部分的论述，你会发现其思想与控制论是暗合的。这样的例子在全书中不胜枚举。

【五】

我必须指出，兼顾 L2 和 L3 层次的技术著作，在我目力所及，不但在架构类书籍当中绝无仅有，在整个技术类图书当中也是凤毛麟角。设计模式可以被认为是系统的局部架构。在 1995 年《设计模式》一书出版之前，实际上有一些面向对象的技术著作在 L2 的层次上分析过产生优秀设计的方法学。我印象最深的是 Andrew Koenig 和 Robert C. Martin 在九十年代初中期的一系列 C++和面向对象著作，提出诸如 handle 和 pimpl 等设计原则。基于这些原则，你可以很自然地在面对问题时设计出漂亮的“模式”。但是《设计模式》出版后，很遗憾的，整个社区将注意力转向罗列设计模式套路并且期望在实践中套用之。这应该说背离了设计模式提出的初衷，也导致设计模式盛极而衰。这是只关注 L1 而 L2 缺位导致的典型后果。

本书作者显然有意识地强调架构认知升级的重要性。在这本以架构为题的书中，作者完全没有罗列“流行架构二十一个”、“你可以套用的架构模式十八掌”之类 L1 的内容，而是几乎将全部篇幅放在 L2 层次上，试图以定义、拟合、类比等方式教授“产生优秀架构”的方法学。这一尝试，无论结果如何，其本身就是雄心勃勃、令人赞赏的。

因此我对于读者阅读此书的建议，一是站在认识论、方法论的层面上与作者对话，二是适当了解一些系统论、控制论的基本思想，这

将有助于理解本书的内容。

我想，毕竟不是每一位读者都像我一样着迷于这些理论的东西，但无论如何，我没有见到过一本介绍系统和软件架构的作品具有如此的认知高度，为此我愿意向有兴趣的读者推荐。

智百科技（AI100）合伙人

孟岩

2017.05.01

序 2：易是变化

> 《大道至易》第 1 版前言节选。

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。

序 3：架构之为物

我用了 20 年的时间，终于有了自己对“程序”的理解——程序是可被组织的元素[①]。这事实上是对程序的可结构化特性的一个阐释，貌似是说着相同的话。然而如果程序是可被组织的，那么“结构化”其实就只是组织的手法之一。这意味着后者——结构化——只是“程序是什么”的一个解，而绝非唯一解。

这就是架构视角的独特处。当它找到一种新的抽象来定义事物时，旧的事物哪怕没有形式与内容上的变化，却在思维框架中有了新的位置、新的理解，以及新的矛盾与冲突。而所谓问题，就来自这些外在视角的变化和内在冲突的产生。架构的目标最终就是直指这些问题，而非解决一个切实的需求，例如写一个程序。

一切的起源在哪里呢？

我将我作为程序员时对“程序”的观察体会写在了《程序原本》这本书中，然而书中最终写到的是一个称为“系统”的东西——它既是一种程序的目标，也是指该目标的规模。那个所谓的系统由一些

[①] 这是我在 SD2C 2016 大会演讲的主题《有前端思想的物联网架构》中提出的观点。

称为“分布、依赖、消息、子系统”等等的基础部件构成。我想在大多数人看来，这些更多地应该是属于架构师讨论的话题集，而非程序员。然而，到了现在你所读的这本《我的架构思想》中，却只剩下了“系统”这个讨论对象，那些基础构件已经全然不见了。

这一切的根源又在哪里呢？

架构本质上是一个映像。洞见映像背后的事实，就如同从镜子去观照现实，知道镜子是一层，知道镜子中的映像是第二层，知道镜子映像所现的实体是第三层。而至第四层时，还要看得到那实体周围的背景，这是实体之为实体所必须的依托，如绿叶之于红花。再深入到第五层，你得知道背景之外不可见的那些影影绰绰的事实，它们是环境中的残片和推想，它们不可确知而又影响着你在镜子中看见的那个主体。再至第六层……

如此层层渐近，才是真正的“镜之用在鉴”，才是“鉴”这一行为的本意。然而一旦你触及到“鉴作为行为的事实存在”，那么你就看到了镜子一侧的自我，进而看到自我之见，看到由自我、镜鉴和自我之见等等所构成的整个系统，这个系统被称为“观察”。当然，在这整个“被称之观察系统”的系统之外，还要有光。否则一切所谓事实都将湮灭，即便存在，亦无可证实，无可证伪。

架构需要那束光来观照事实，以证明自己的存在。

周爱民

2017.02.26

关于本书

> 节选自《大道至易——实践者的思想》（第二版）。

从现在开始，原书（《大道至易》第一版）被分成了三本书，以电子书的形式由图灵出版公司发行。这些书采用了免费共享的授权（是的，你可以很自由地分发它），分别面向各自己的读者群体。它们不但在内容上独立清晰，而且在行文上各持风格，再也不受原书的影响——例如我恢复了所有章节、小节的标题，分别设计了它们的封面等等。

这三本书分别是：

- 《大道至易——实践者的思想》（第二版）

  面向软件工程相关角色。

- 《程序原本》

  面向程序员、软件设计师。

- 《我的架构思想——基本模型、理论与原则》

  本书。面向架构师。

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。

感谢那些曾予我帮助的人们。

感谢 Joy。En，……我最爱的妻。

目录

引言：架构师的思维

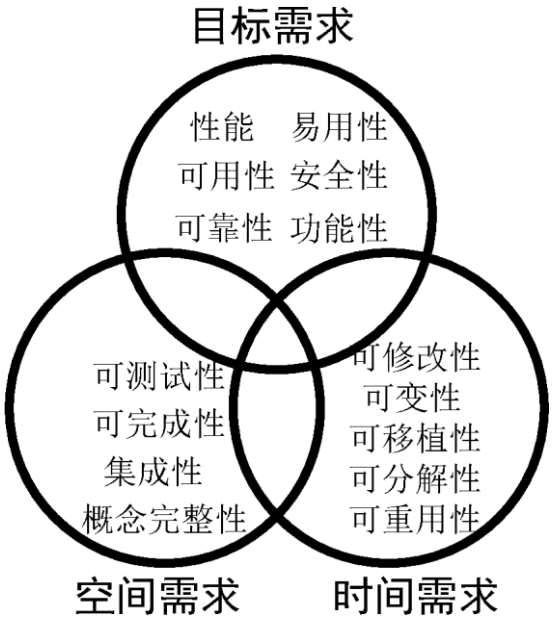
一个人太不切实际，我们称之好高骛远；一个人有眼界视野，我们称之高瞻远瞩。同样是高、远，为何描述着两种完全不同的人？因为前者的好、骛讲的都是追求，而后者的瞻、瞩，指的却是具体的行动。当我们把高远的目标只作为一种追求，而不付诸于实践的时候，就是不切实际；当我们把它变成“时时顾看”这样的行动时，我们就渐渐地变得有眼界视野了。

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 1 所示[①] [②]。

图 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] 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。

(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 1 所示。

表 1 架构师在思维过程中使用的工具

	行　　为	思　维
1	语言文字与架构思维的比拟	系统映射
2	对一个确定背景下的高、远的明确定义	系统概念
3	探求上述概念对既有的知识的包容与冲突	系统范围
4	确定为架构产出	命题与设问
5	既然上述是架构产出，则上述过程为架构思维的过程	论证与求解
6	重复这一过程，以检验架构思维的有效性	验证

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 1* 与*表 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。

编一：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。本编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。本编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。

本编所讨论的系统是一个泛义的概念，并不具备规模性的含义。同样的原因，本编所讨论的架构也是泛义的。

第 1 章 了解系统的过程

1.1 感受一个系统的事实

我常常设想一个场景，这个场景是如此的简单，以致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。

1.2 系统是一种认知，而非分析的结果

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，更是一种外界——之于这个系统——的认知[①]。

1.3 认知理论中的知识：知得与识得

认识与感受是不同的，我们接下来讨论“认识”。

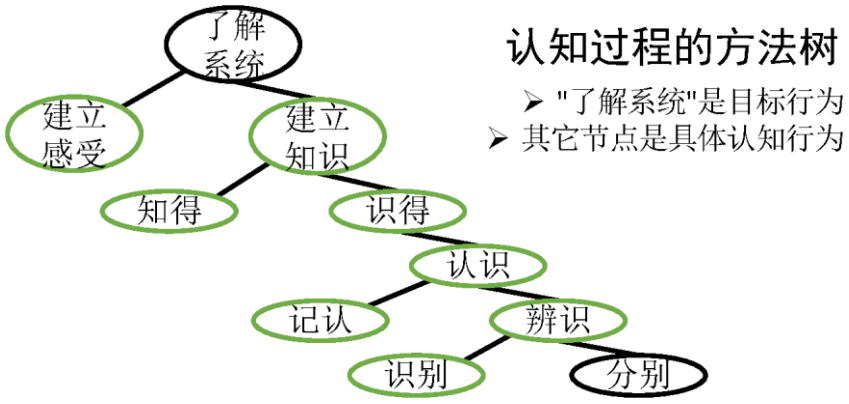
一个人认识到另一事物，包括对这一事物的两种了解，其一是它的外在，其二是它的内在。例如说认识一个人，男女老少、发肤形貌等这些是外在特性，品性德行、气质教养等这些则是内在特性。“认识”的这两点要求，无论要了解的对象是复杂的人还是简单的石头，都是必要的。

我们获取知识有两种手段，是所谓“知得”与“识得”。在《程序

[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。

原本》一书中讨论**抽象**这一主题时，我们提出过这两种手段[①]：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[②]，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。与这一切相关的概念，可以描述在下图所示的方法树上（这里引用的是后续文字中的图例，所以图序号有异）：

图 5 认知过程的方法树



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，我们后来在《程序原本》一书中讨论到的 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

① 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。

② 不一定是指“触摸”，可能是看见、听见、嗅见等多种接触形式。

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。

1.4 尝试一个“建立知识”的过程

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。

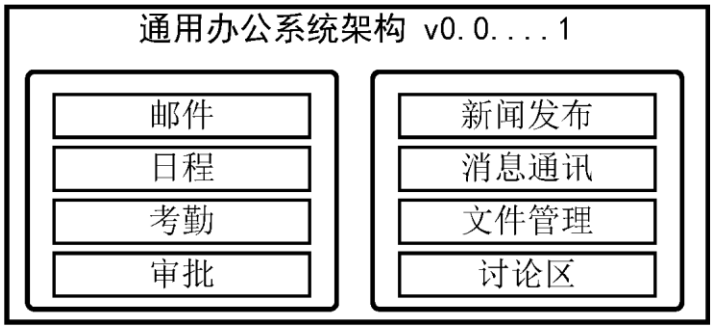
这几点事实显而易见，是由系统本身决定的[①]。我们可以因此找到一些系统的组成部分：

(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 2 所示。

图 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] 事实上并不尽然，我只是有意地忽视了这一过程的复杂性。

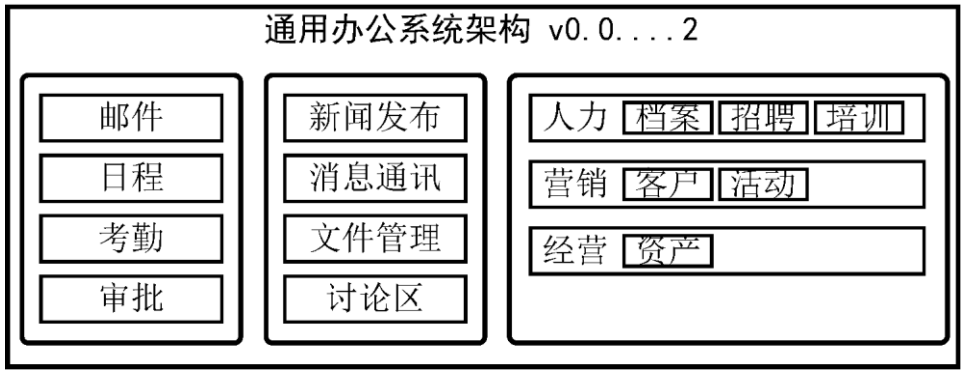
(1) 人力，即档案、招聘、培训；

(2) 营销，即客户、活动；

(3) 经营，即资产。

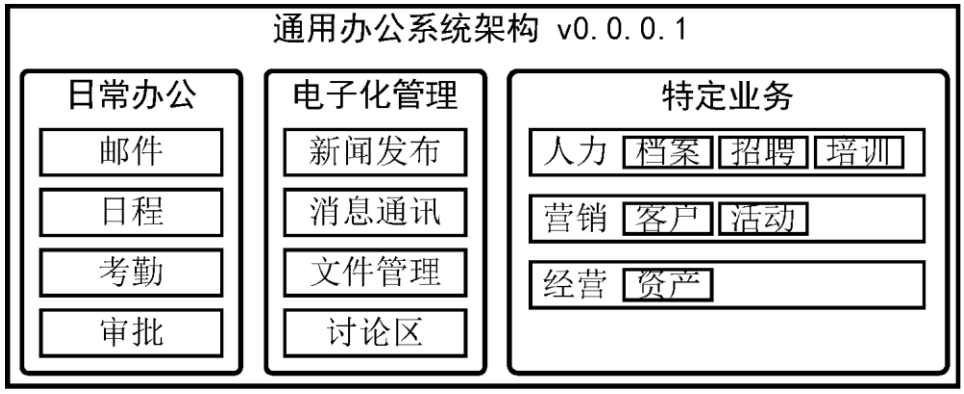
据此我们进一步补充这一系统的架构，如图 3 所示：

图 3 通用办公系统架构 v0.0.…2



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”，如图 4 所示：

图 4 通用办公系统架构 v0.0.0.1



几幅架构图的演进关系并不难理解，但有一点点差异：图 2 至图 3 的标题中的版本号是“v0.0….x”，而在图 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 却可以称为“一个最最最初级的架构版本”？

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？

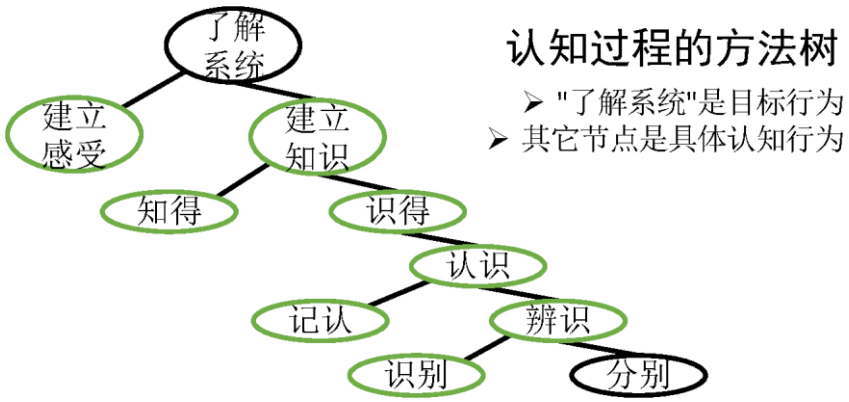
又或者问：你何以在系统中做出一些设定，而非仅仅陈述现实系统

的事实？

1.5 “建立知识以陈述现实系统”是不足以架构系统的

如前所述，了解系统的一些具体方法，大体来说类似于图 5 所示的一个认知过程的方法树。

图 5 认知过程的方法树



我们事实上只讨论了认知行为中很小的一个部分[①]。**“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

[①] 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。

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。

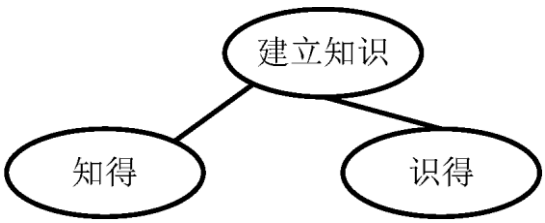
第 2 章 知识的构建

2.1 观察者的背景差异带来了更多不同的正确映像

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 6 中的“知得”而非“识得”。

图 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。

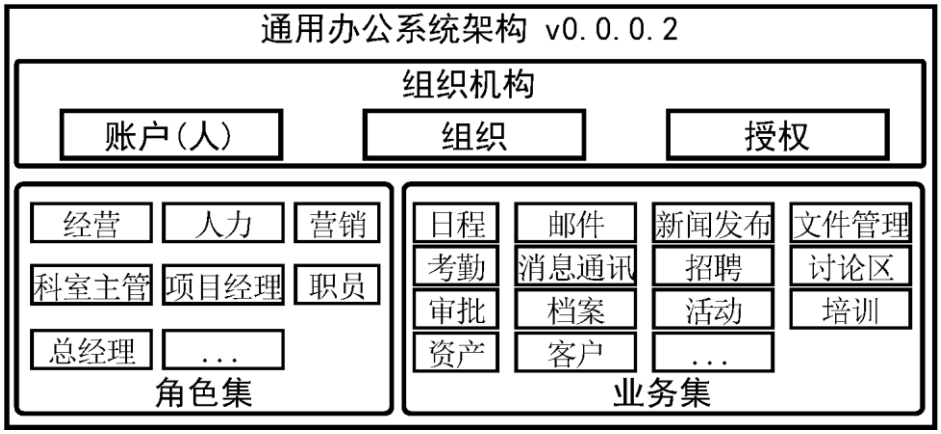
- 一方面总经理很忙，他只能只言片语地向你介绍他的具体工作，因此你可能需要找到他的秘书来协助整理这些需求。而一旦主角由“他”变成了“他的秘书”，活动由主动介绍变成了侧面观察，“总经理在办公系统中的需求”就变得并不那么可信了。
- 另一方面办公系统本身就存在多面性，例如它既是每个人的私人工作平台，又是多个人的公共协作平台，还是系统资源的承载平台……那么你观察这个系统的时候，得到的信息将会是向多个方向发散的，以至于你会觉得：向任何一个方向“多做一些”都会变成一个巨大的分支。

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 7 所示的样子。

图 7 通用办公系统架构 v0.0.0.2



2.2 这种差异表现了不同的“架构意图”

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[①]。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[②]。

“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；

[①] 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。

[②] 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。

- 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[①]；
- 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，我们可以认为，“架构 v0.0.0.1”完成的是**办公（功能）系统**，而“架构 v0.0.0.2”体现的架构方向应该被称为**办公管理系统**。关键的区别在于，前者仅仅是对现实系统的一些事实的复制，而后者体现了一种架构意图。然而一旦架构思想中出现了这一意图，

[①] 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。

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？

2.3 抽象概念与模型是展示架构意图的方式之一

真实的情况通常是这样的：客户提出“办公系统”时，并没有打算开发寄予了管理期望的一个软件产品。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：

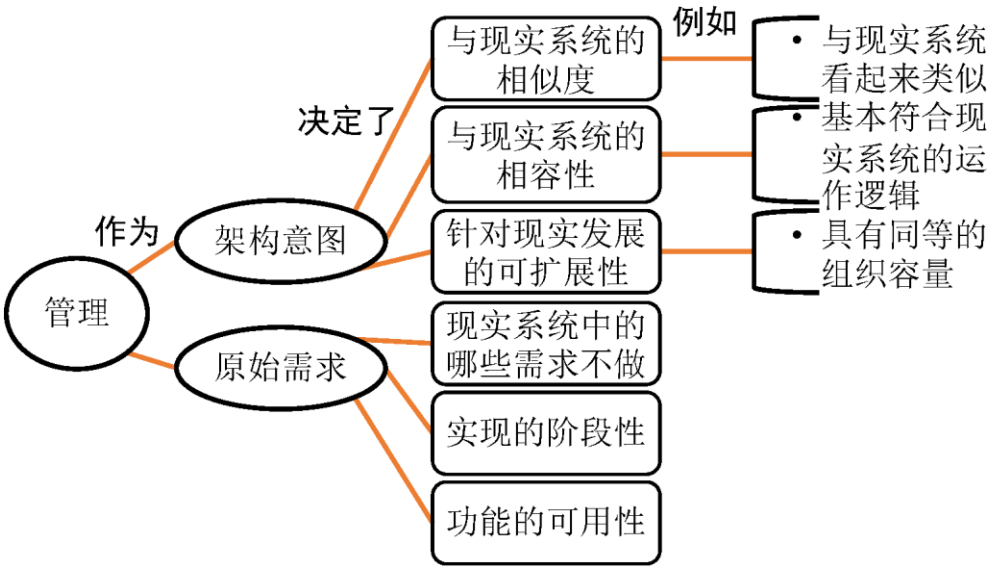
- 与现实系统看起来类似的
- 具有同等的组织容量的
- 基本符合现实系统的运作逻辑的

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 8 所示[①]。

> 图 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

[①] 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。



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“具有同等的组织容量”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。

2.4 系统的识得，是在架构意图的逐步清晰中渐行渐显的

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[①]；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？

[①] 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。

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[①]，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**”[②]以及“被授权者是可以行使系统行为的**角色**”[③]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 2 是一个简要的考察。

表 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] 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。

[②] 它决定了逻辑“*if Auth.isManager(User1) then Auth.assign(User1, User2, 'aAuthDescription')*”的合理性。

[③] 它决定了逻辑“*if Auth.has(User2, 'aAuthDescription') then doSomething(…)*”的合理性。

架构话题	收　　益
可测试性	组织结构及其内部的授权关系可以与现实系统互为验证，并且可以通过授权接口对任何一种权限所涉功能以及权限之间的组合关系加以测试——而不影响系统的其他部分
可完成性	基于成熟的树型结构抽象及相关算法，可表达组织机构的层级关系，以及部门与成员的辖属关系
集成性	通过赋以用户多种角色以使用户具有丰富的操作能力；可通过叠加与角色对应的功能模块来扩展系统
概念完整性	通过**授权**建立**组织机构**，通过**验证授权**决定对**功能**的使用，通过功能的叠加与依赖关系来**组织系统**

（续表）

架构话题	收　　益
可修改性	系统功能依赖授权接口Auth.has()来组织，在has()接口不发生明显变化的情况下，系统任何部分的修改都不会对全局产生影响
可变性	功能叠加或变更与组织机构无关，反之亦然
可移植性	权限与角色集具有很强的可移植性
可分解性	组织机构、角色集与功能集各自独立
可重用性	除功能集外，其他部分是可以跨系统重用的；功能集在对当前系统的重构以及持续开发中是可重用的

注：对于表中“可移植性”的进一步说明：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。

2.5 知得，始于抽象概念的构建之后

“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[①]，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。

[①] 严格地说，“管理”这一架构意图的由来是我们在小节“2.3 抽象概念与模型是展示架构意图的方式之一”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。

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[①]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[②]中所出现的概念：管理、组织机构、授权、

[①] “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。

[②] 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

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；

在“正确地架构一个系统”这一方向上有着相当可喜的表现。

- ……

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。

2.6 “识别架构意图”的核心理论与方法

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[①]或意图[②]。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。

---

① **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。

② 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。

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[①]；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[②]。

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[③]，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。

[①] 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**。

[②] 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。

[③] 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组识别用的凭据 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。

- 流转关系[①]：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[②]：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[③]，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[④]，则账户 A 和

① 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。

② 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。

③ 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。

④ 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。

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[①]。仍以上面的支付系统为例，我们假定[②]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[③]。总体来说，任何一个架

---

① 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。

② 该假定是以“需要实现为开放系统”这一动向为基础的。

③ 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？

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[①]：

> 一个跨领域的开放支付平台。

[①] 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。

第 3 章 最初的事实

3.1 真相是相，而不是真

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[①]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。

(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。

质疑那些主观判断，是系统架构思维中的第一步。

3.2 如何推翻那些最初设定的“事实”？

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。只有将第一点和第二点事实合而为一的时候[①]，才能表达如下的一般过程[②]：

> 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

[①] 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，是对第一点事实的补充。

[②] 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。

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[①]；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[②]，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

[①] 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。

[②] 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。

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；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[①]；
- 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。

[①] 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、面向内部系统的开放接口。

3.3 仰首者瞻，凝神者瞩

在讨论中，我们的下面两点设定将会带来一个“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[①]。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

[①] 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，甚至是 CEO/CTO 本人。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。

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[①]”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

[①] 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。

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[①]，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。

3.4 找到问题也就等于找到了解

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[②]？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

[①] 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。

[②] 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。

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 9 所展示的。

图 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[①]。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？[②]

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”

[①] 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。

[②] 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。

的需求呢？可见，我在这后半段中所经历的，其实是图 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，也就同时看到了那个解的抽象。

图 10 面向问题的思维活动

煮一枚鸡蛋[①]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

[①] 这是在《程序原本》“第 18 章 系统”中构想的一个思维问题。

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”[①]——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[②]，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。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。

---

① 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）。

② 参考温伯格《你的灯亮着吗？》。

3.5 反思那些事实与问题

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[①]。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[②]：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”[③]的应用与设问。

[①] 这是在《程序原本》“第 8 章 执行体与它在执行过程中的环境”中提出的问题。

[②] 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。

[③] 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。

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。

编二：架构是过程，而非结果

> 本编讨论的“系统”与上一编并不相同。这里的系统是一个规模用词，因此本编将基于“领域集”的概念来讨论系统问题。出于这一点设定，本编也提出了“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，并且将该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。
>
> 本编中第 5 章主要面向形成论讨论，第 6 章主要面向组成论讨论。

第 4 章 架构师的能力结构

4.1 组织视角下的架构师角色

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。

因此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，所以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。

4.2 架构师的能力模型

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，

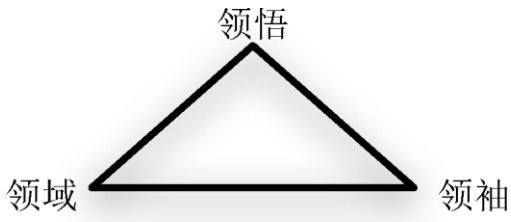
---

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。

我认为他们——架构师应当具备的能力包括图 11 所示三个方面[①]。

图 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[②]：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[③]。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[④]——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（即领导能力）主要是在组织视角下对管理者

① 其中，（与计算机无关的）某些领域细节是不需要在这里讨论的。

② 参见本书“引言：架构师的思维”。

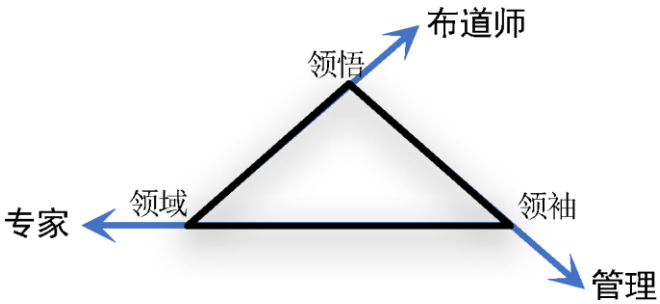
③ 如果架构是一个“决策”的过程的话，多样性就是必须的。

④ 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。

（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[①]，而是指架构角色所形成的、超出组织结构的影响力[②]，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 12 所示。

图 12 个人能力取向与职业倾向的对比



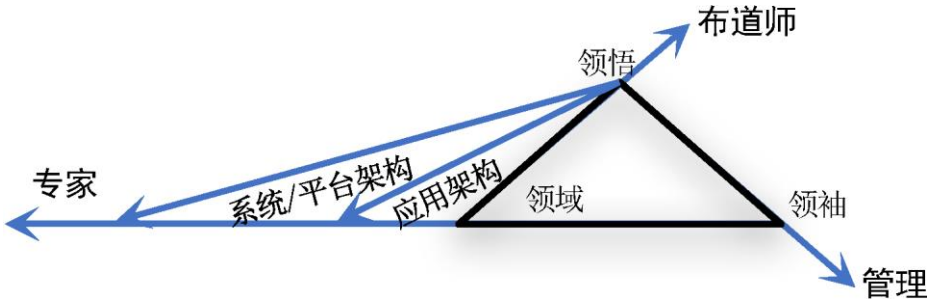
当从领域专家这一方向上衡量时，在领域方面的背景知识（一定程度上）反映了他可以应对的架构的规模——之所以选择从领域方面

[①] 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。

[②] 参考本书附录之“附三：超越软件架构——组织与架构”。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。

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[①][②]，如图 13 所示。

图 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 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”[③]为例，他应当具备与之平衡的领悟与领袖能力

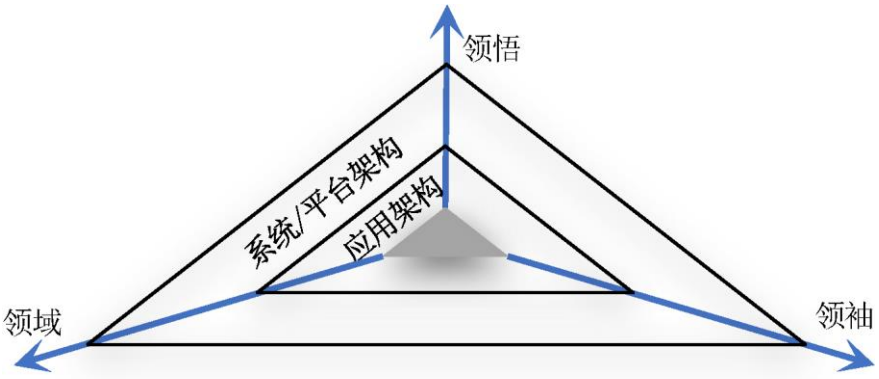
① 仅软件开发领域而言，架构师的领域能力与他能应对的开发规模是相关的。参考《程序原本》一书自“第 5 章 从功能到系统”开始论及的四种开发规模：技术架构对应于功能（function)与程序（program），应用架构对应于应用（application），而系统/平台架构对应于系统（system）。

② 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。

③ 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。

（如图 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 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[①]，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[②]。

4.3 架构决策

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

[①] 参见本书附录之“附三：超越软件架构——组织与架构”中所讨论的 VEO 模型。

[②] 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。

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。

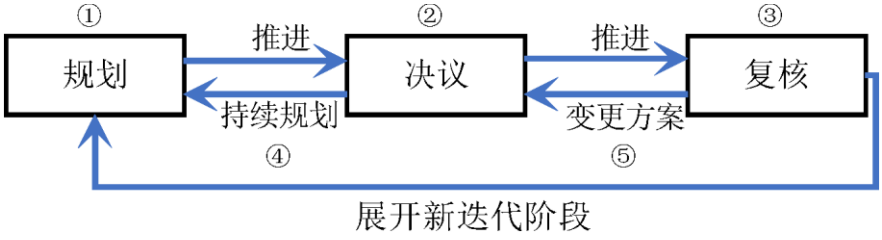
另外，参考格拉斯问题解决模型[①]，整个架构推进过程还涉及两个时间方面的决策：

(4) 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；

(5) 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[②]”上所需要的决策过程。其基本模型如图 15 所示。

图 15 在“架构整体”上所需要的决策过程

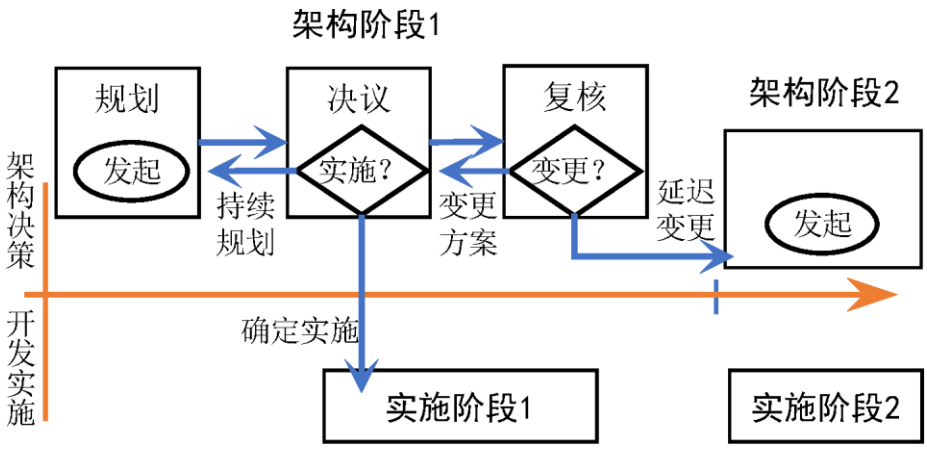


而图 16 则用于说明将该模型应用于一个开发实施场景中的具体架构决策过程。

图 16 多个实施阶段下的架构决策过程

① 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。

② 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。



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。

4.4 有价值的决策是对意图的响应

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[①]。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。

[①] 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。

第 5 章 系统架构与决策

5.1 系统架构的提出

“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？

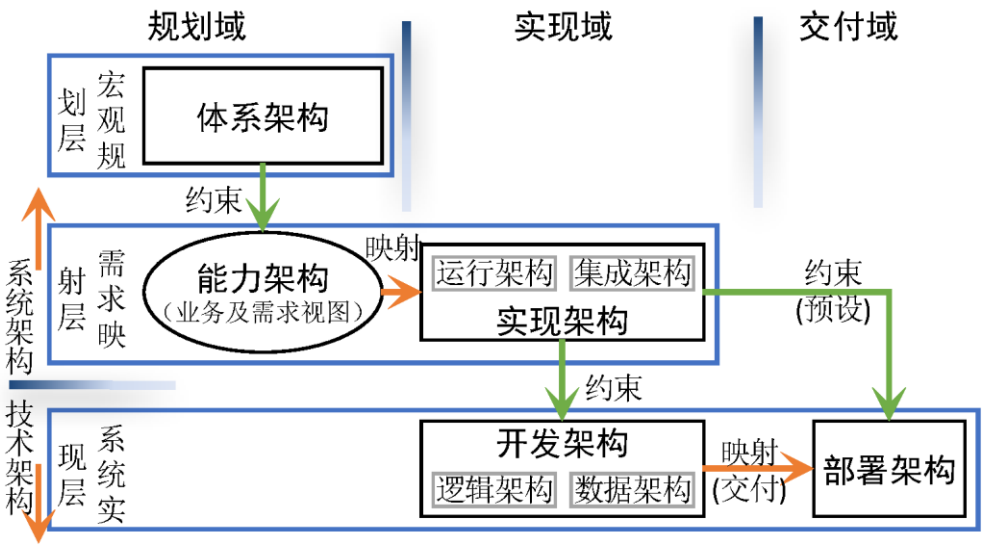
在继续讨论之前我必须再次强调：接下来的讨论并不基于该系统架构之局部（例如子系统/应用的架构），而是面向其本身——**系统架构**。[①]

[①] 更具体来说，后续的 5.2~5.4 小节讨论上述的设问(1)，而 5.5~5.7 小节分别讨论设问(2)~(4)。

5.2 形成论：参考模型 M0 以及可参照的示例

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，即是小节“4.3 架构决策”中基于架构师团队所讨论的决策过程[①]。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 17 所示。

图 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，其原因是它们对部署与开发的约束作用最为明显。

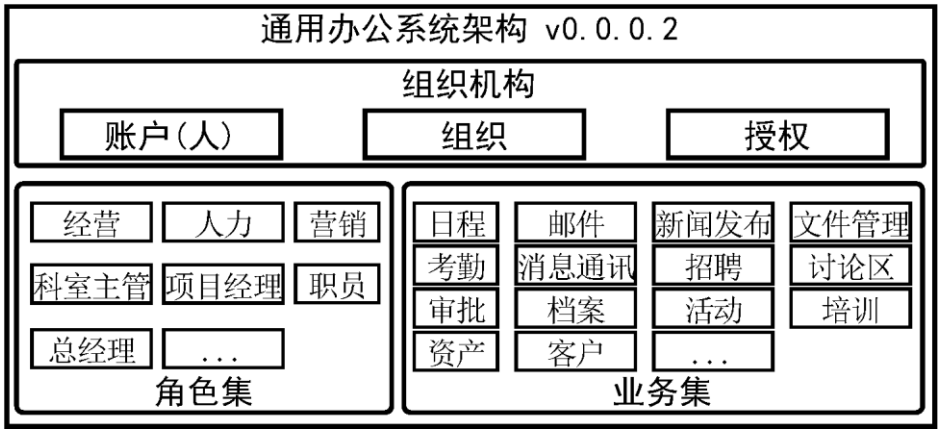
① 本书只讨论了决策过程模型，但并不否认团队推动架构的其他过程方法。

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 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 18 中）。确切地说：它充其量只能算功能架构[①]，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。

图 18 通用办公系统架构 v0.0.0.2



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

[①] 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 18 中体现。

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）。

5.3 参考模型 M0：细解各部分的形成过程与关系

现在，让我们再前进一步[①]，将这一架构意图表达为一个概要的系统架构的模型[②]，如图 19 所示。

> 图 19 一个概要的系统架构的模型

[①] 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。

[②] 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。

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[①]。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）[②]来表达，例如将不同的功能包装并发布成服务，如图 20 所示。在图 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[③]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。

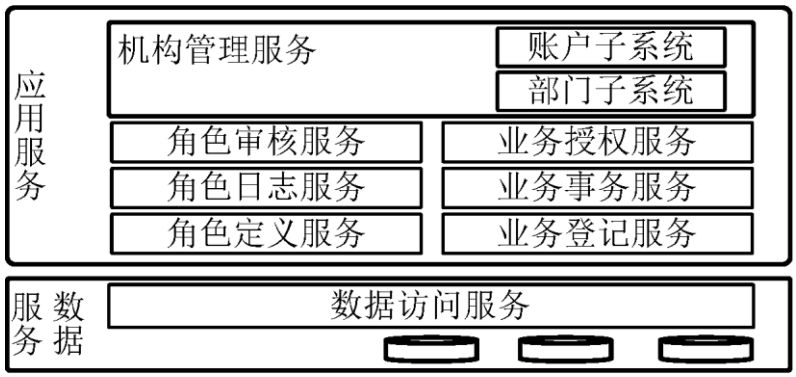
---

① 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。

② 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）包或组件。

③ 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。

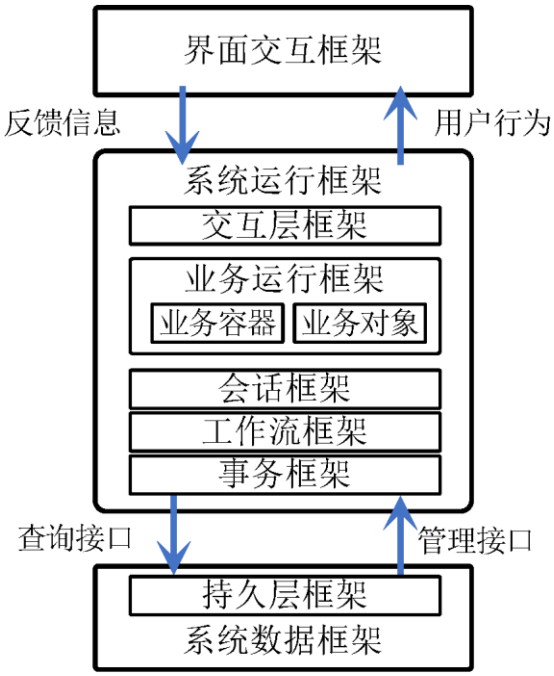
图 20 服务层的模型



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层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 21 所示的架构。

图 21 框架层的模型

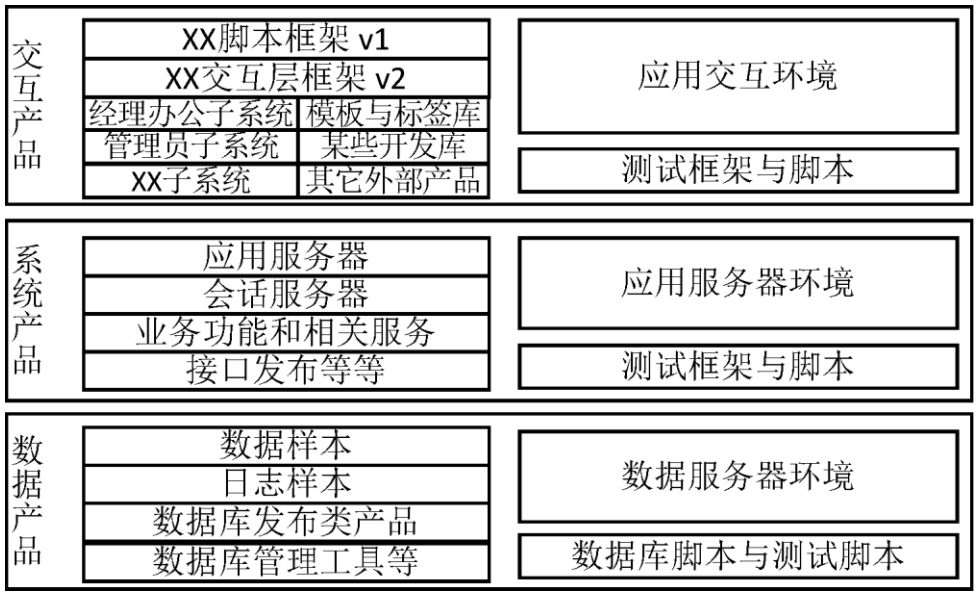


注意框架的“可运行特性”，这意味着框架必须能描述数据和（阶段性地）持有数据的逻辑之间的关系。上述架构中使用了箭头，用以指示某些信息流的流向（call）与转换（transation）。我们必须保证上述框架层在数据与逻辑关系上是完整的、足具的，否则框架本身便无法实现“动态的运行架构”这一目标。

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 22 所示的产品层的模型表达。

图 22 产品层的模型（面向产品的集成架构）



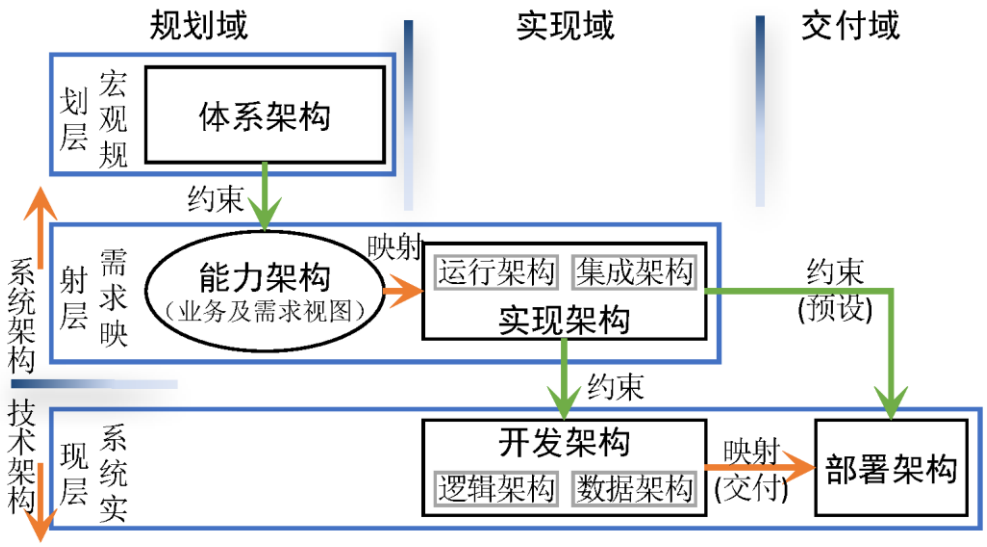
这个架构表达了维持框架层正确运行所需的环境、工具与测试脚本等部件。这些内容/部件中的一部分是开发活动所需产出的产品，一部分是第三方产品，并且一般来说后者是首选。大多数情况下，集成架构表达的是需要集成的产品内容及其之间的关系。此外，因为集成架构的内容是产品而非程序的调用接口，所以它们之间的关系将主要是组合、依赖与层次等，而非调用或数据返回。

5.4 “通过什么来影响什么”作为一般过程是可行的，但不完备

通过产品、框架、服务与功能这样的层次，上述系统架构整体地表达了它作为“实现架构”对实现域和交付域（以及阶段）可能构成

的影响。对照图 17（这里引用的是此前文字中的图例，所以图序号有异）所述的形成模型 M0，可以看到在该系统架构中：

图 17 一个可参考的架构形成模型 M0



- 功能架构：它是实现架构中的组成部分，把由能力架构映射而来的能力分割为基本独立的**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，图 19 所示的系统架构可以通过 M0 模型展示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：

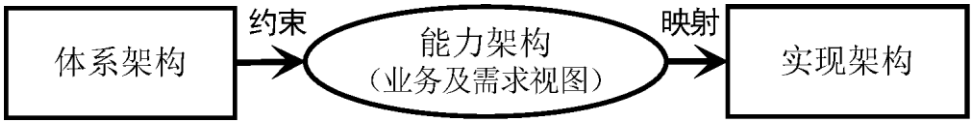
> 一个以企业管理应用为目的、以业务功能为核心的三层架构。

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[①]。

最后，我们对“架构形式模型 M0”作一些简化，就可以得到图 23 所示的系统架构的一般过程。

[①] 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。

图 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[①]。

5.5 平台与框架的极致是“做到看不见”

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。

[①] 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。

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[①]。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

[①] 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。

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。

5.6 层次结构是架构的一种平台化表现方法，而非架构本身

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 MO”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[①]”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[②]，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

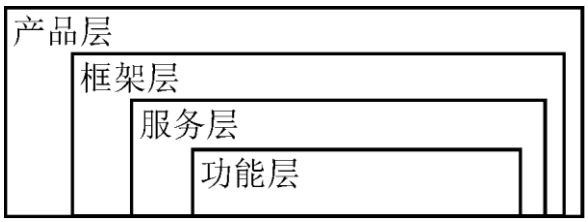
[①] 架构为什么要分层以及如何理解架构的分层，是下一章讨论的内容。这里所谓的**三层架构**或**多层架构**这样的模型，是讨论中用以参考的、现实中的例子而已。

[②] 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。

化地做出两点假设：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 19 所示（重复于图 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。

图 24 一个概要的系统架构的模型



对于一个基于该模型构建的系统，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：

- 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[①]：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。

[①] 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。

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。

5.7 形成论的另一种求解：架构规划

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 3 所示。

表 3 架构规划在不同阶段中的重点

阶段	单一产品	特定领域产品	通用领域产品或作为产业链环节的平台	跨 平 台
重点	☐快速实现功能	☐可预测的定制性 ☐产品化 ☐稳定性 ☐快速集成与完整的测试	☐平台化 ☐非常强的产品定制功能 ☐高可靠性，以及降低错误处理成本 ☐丰富的接口，以应对基于平台的内部与外部开发	☐接口标准化 ☐三方开发模式 ☐战略推进与系统规划如何同步的问题

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，我们可能将当前系统的目标设定为：

> **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 给出了一个示例。

表 4 架构意图在不同阶段中的变化与入手点

阶段	单一产品	特定领域产品	通用领域产品或作为产业链环节的平台	跨平台
架构意图	宜于快速实现的功能性架构	宜于定制的、以功能插件和插件框架为核心的架构	面向产业链核心数据的平台	跨平台产品簇
启动条件	客户需求	获得三家以上的典型客户	涉及两个以上的行业	跨终端的版本
入手点		既有功能的一般运行过程，基于此形成插件框架；核心概念的提取与抽象，并据此形成插件规格与调度原则	开发新框架以整合现有框架的运行过程；提取核心数据概念，并讨论其中核心的产业模式，以及可能的变化	接口的标准化
周期	1年	2年	2年	2年

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 6 章 架构的表达与逻辑

6.1 从暗示、隐喻，到抽象概念的表达

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”的人心里多少要犹豫一下；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，我们前面在讨论“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。

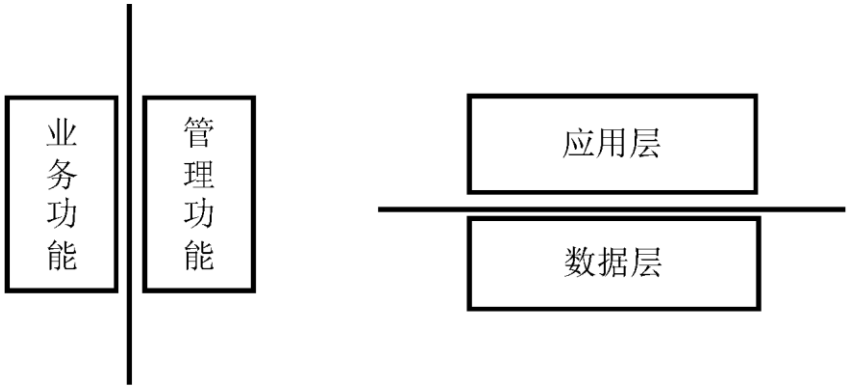
但是这并不妨碍我们要求对架构的表达必须清晰准确[①]。因为实施过程必将依赖架构的结果，亦即是架构的最终表达与决策。**架构表达**是在架构过程的阶段性成果的基础上所进行的、尽可能准确而清晰的叙述。如果它不能尽量准确地反映架构过程的阶段性成果，那么它也就不能作为下一个阶段（无论是实施还是新的架构迭代）的有效依据。换言之，如果对架构结果的表达是模糊的，则该结果是无意义的。

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？

6.2 理解线与线框

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 25 所示。

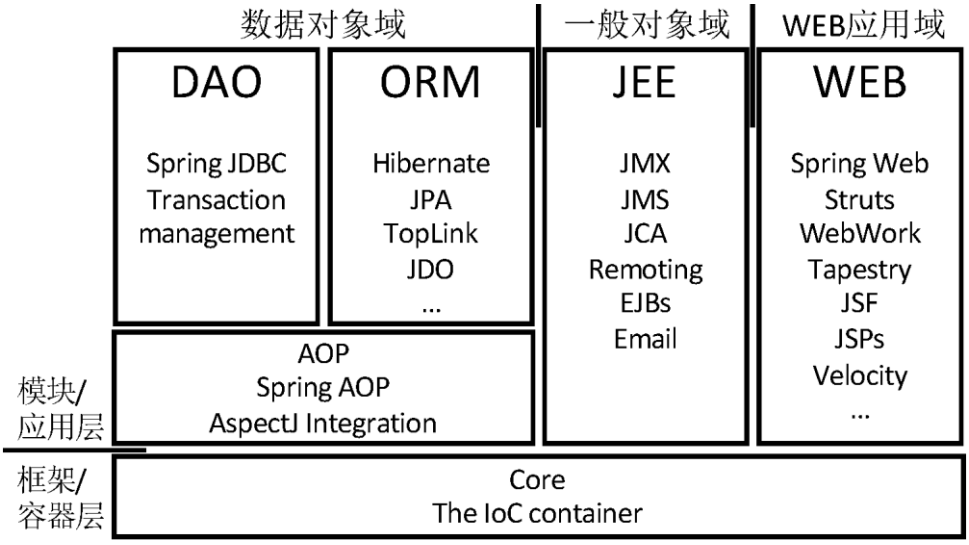
图 25 整体划分成两部分



① 《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”。

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 26 所示的“Spring 架构图”中隐含的一些领域与层次信息。

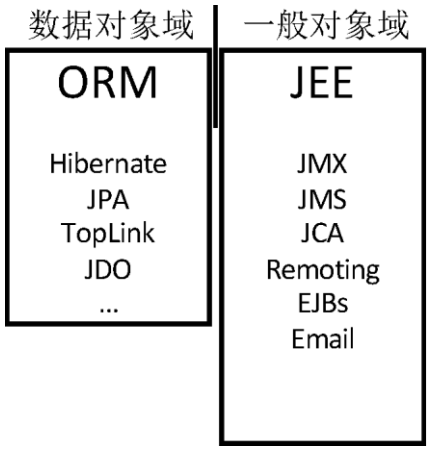
图 26 Spring 架构图



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 27 中的“数据对象域”与“一般对象域”是二者的分类依据[①]，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。

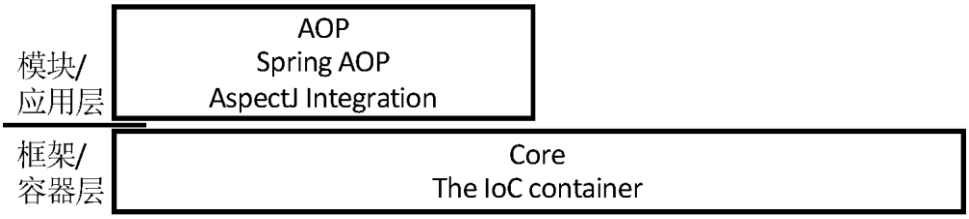
[①] 它表明的其实是“普遍/特殊”这样的分类法。

图 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 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对其外围作**“支撑”**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[①]。

图 28 Spring 架构图：AOP 与 Core 之间的层次与分类依据



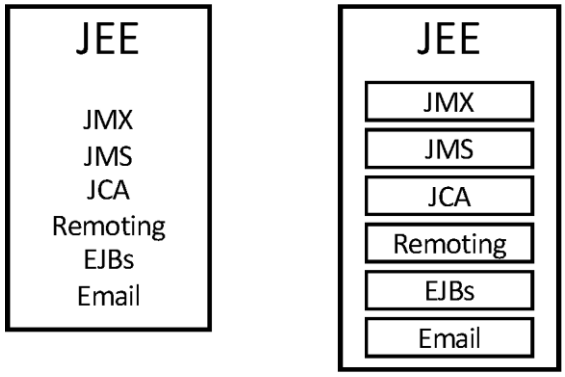
如上，我们得到了一个整体的各个部分：相互间可以用线条来区隔，

[①] 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。

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 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 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。

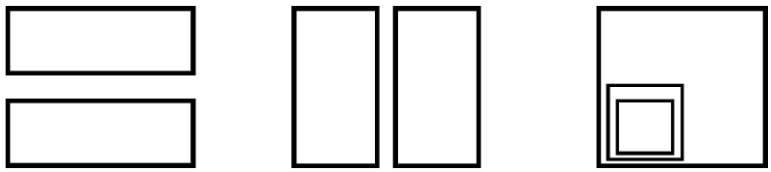
6.3 对系统或其构件的不变性的表达：平台、框架与库

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 30 所示的三种结构。

图 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[①]。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，那么它适合用层次结构来表示；
- 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

[①] 在“第 5 章 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。

来说，它带来的复杂度应是衡为“1”的[①]；它决定了系统整体的性状是不变的[②]。

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[③]，其领域总量倾向于不变；嵌套结构适宜构建**框架**（framework）的过程[④]，其核心领域（或核心过程）是倾向于不变的。

6.4 系统总量不变，其本质是复杂性的不变

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

---

① 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”之间是存有区别的。

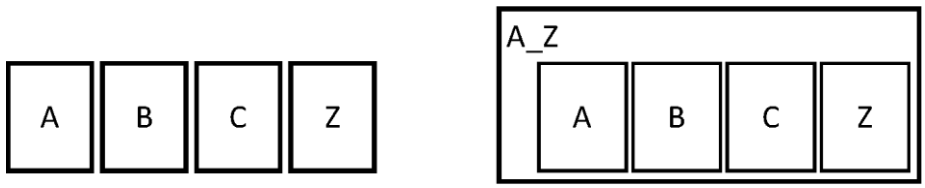
② 类似这种“**性**（状）”的不变，并列与嵌套结构表达的分别是系统的“（总）**量**”与“（本）**质**”上的不变。

③ 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。

④ 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。

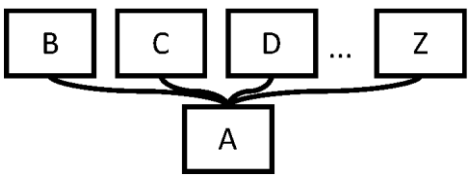
（的所有域）是可以被视为一个整体的。换言之，拆分或不拆分，只是对规模的分解而不会导致关系或相关处理的增减。就系统整体来说，其规模将因 A..Z 的个数的增加而线性增长，但其复杂性仍然是衡为 1 的。因此，图 31 所示的两个图例是等义的。

图 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 32 所示形式的结构。

图 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 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 33 所示的嵌套结构。

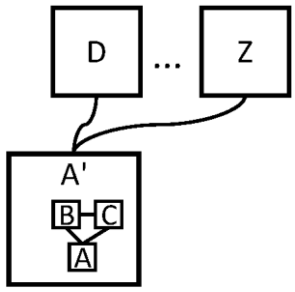
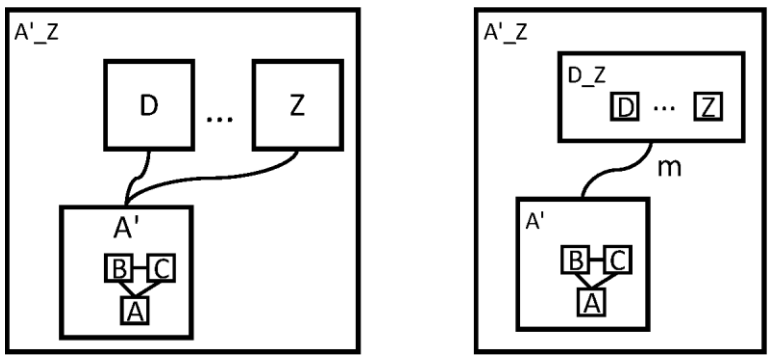


图 33 以 B_C 之间存在关系为例改写架构的表达

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 34 所示。

图 34 嵌套结构与并列结构

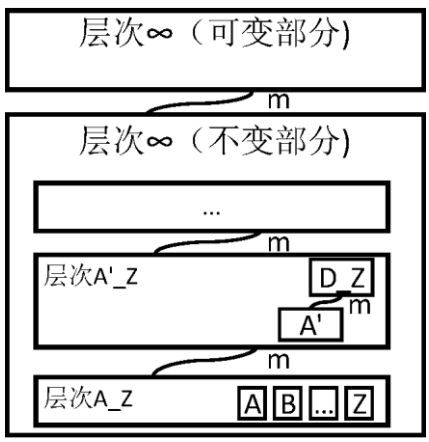


综合上述有关嵌套结构（A’_Z）与并列结构（A_Z）的讨论：既然它

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 35 为例，也就是说：

- 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[①]，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。

图 35 其他结构在层次结构中的含义



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，

[①] 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。

也决定了“不变部分的复杂度是 1”的单位大小。

6.5 化繁为简：控制架构的复杂性

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。

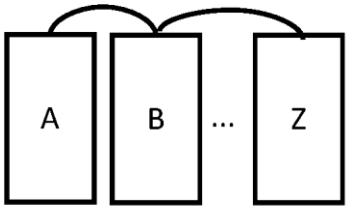
所幸，这些可被讨论的关系是有限的。这在《程序原本》一书“第 18 章 系统”中就曾经提出过[①]：任意领域的系统对“当前系统”的需要只有两个，亦即寻求计算资源，或寻求数据资源。回顾此前的讨论，若 A 需要 B 的数据资源，我们称为**数据依赖**；若 A 需要 B 的计算资源，我们称为**逻辑依赖**；若将 A 和 B 都视为逻辑（亦即是可计算的资源），并讨论二者之间的关系，那么我们就会看到**（逻辑或数据的）时序依赖**。

这些“有限的关系”的复杂性 m 如何呢？

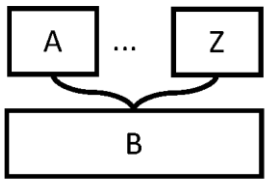
首先，并列结构之间是不应存在关系的——这与我们此前设定的并列结构的表达原则有关。如果它们所表达的领域之间的确存在依赖（如图 36a 所示），那么事实上可以将被依赖域下沉为一个独立层次（图 36b），并使其他域基于该层以使整体表达为层间（向下）依赖关系（图 36c）。

图 36 并列结构之间的依赖

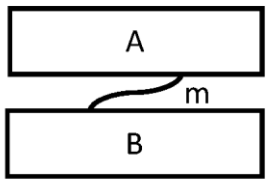
[①] 参见《程序原本》第“18.4 聚焦领域之于系统的主要需求：维护状态或接受消息”小节。



图a：并列结构的
被依赖域（假定）



图b：该域下沉
为被依赖层



图c：其它域可
以理解为一个层

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 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[①]。

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[②]。

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。

[①] 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。

[②] 进而也会破坏由确定性带来的稳定性，持续性等架构特性。

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[①]：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 37 所示。

图 37 将领域转置为层次带来的依赖关系

图a：相互依赖的领域　　图b：转置导致的层次依赖

我们先讨论两种时序依赖关系之一[②]，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[③]，如图 38 所示；或将 A 和 B 的数据

① 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。

② 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。

③ 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。

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[①]。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 38 层间依赖：对数据时序依赖关系的分析与解构

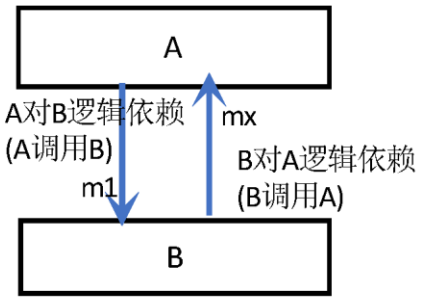
图a：数据相互依赖　　图b：抽取A的数据至B

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[②]。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 39 所示。

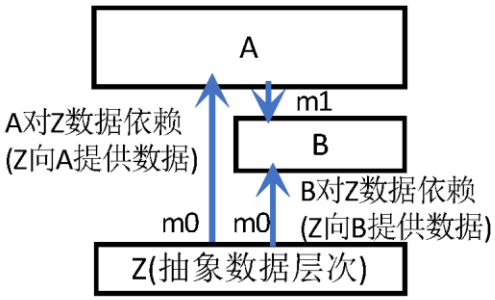
图 39 层间依赖：对逻辑时序依赖关系的分析与解构

[①] 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。

[②] 参见《程序原本》一书“第 16 章 依赖”。



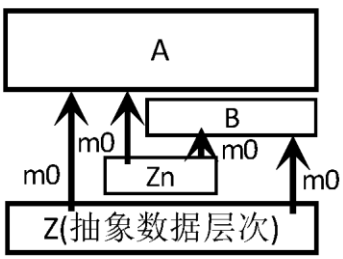
图a：逻辑相互依赖



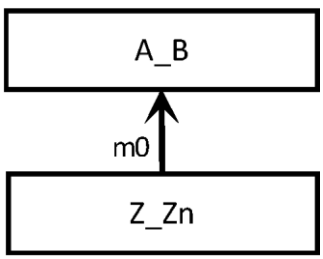
图b：添加新的抽象数据层次

可见这两种情况（图 38a 与图 39a）中产生的向上依赖 mx 都是可以被消化掉的。并且，对于图 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 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 40b 所示）。

图 40 层间依赖：归并



图a：对m1的处理



图b：对层次的归并

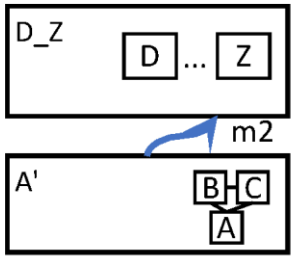
由此得出的进一步的推论是，上层对下层“可以”仅有确定的数据依赖关系：下层对上层的调用，总是可以通过数据的下沉来避免；多层之间向下的调用，可以通过公共数据层来避免。这是一种理想

状况[①]，也是“逻辑－数据层”这种两层基本结构的由来。

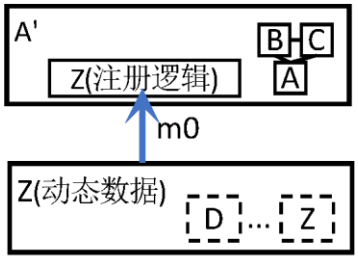
然而还有最后一种情况，会对这种理想化的层次结构设计造成一定的冲击。这种情况发生在将嵌套结构转化为层次结构时，其“核心部分”必然位于底层——因为它是可确定的、可预先确知的。如此一来，在这一层中必然存在底层向上层的关系，如图 41a 所示。

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[②]，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 41b 所示。

图 41 层间依赖：对嵌套结构转化为层次结构的处理



图a：嵌套结构层次化导致的向上调用



图b：对动态数据的注册逻辑

① 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。

② 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《程序原本》一书“第 3 章 抽象”。

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[①]。

6.6 系统确定性是界面原则的核心

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然面临我们持续开发行为将会带来的影响。

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[②]。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；

---

① 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。

② 在 Walter F. Tichy 在 ICSE 1992 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。”

- 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[①]。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 Int_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。

[①] 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。

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[①]；
- 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[②]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[③]。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[④]。

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。

---

① 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。

② 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。

③ 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。

④ 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。

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 5 列举并分类 Erlang 的一些实践性原则[①]。

表 5 三类界面原则的示例：Erlang 的一些实践性原则

类　别	原　　则	注
系统原则	自顶向下	
	不要（从最初就开始）优化代码	
	“最小惊讶”原则	*
模块原则	尝试减少模块间的依赖	
	将公共代码放到库中	
	将“绝招”或“脏”的代码隔离到独立模块中	
	使用设备驱动隔离硬件接口	

（续表）

① 引自“Sofrware(SW) Engineering Principles”，出自 Erlang 项目的公开文档“Erlang Programming Rules”。

类　别	原　　则	注
表现原则	尽可能少地从模块中导出函数	
	不要假设一个函数为何被调用或调用者如何使用返回值	
	尝试消除副作用	
	禁止私有数据结构泄漏到模块全局	**
	保障代码确定性	
	不必要的“防御性编程”	***
	在同一个函数中完成与撤销（例如open/close）	

> * 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[①]，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] 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”。

编三：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：它能够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 7 章 架构原则

7.1 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。因此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，进而出现“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。换个角度，只有当二者站

到“共同解决问题”的角度上来看，才是共赢的，进而问题本身就变成了焦点：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

[①] 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。

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，则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[①]。

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

[①] 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。

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。

7.2 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[①]。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[②]。

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

[①] 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。

[②] 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。

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[①]。

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[②]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[③]，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，

---

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 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。

③ 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。

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”[①]。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。

7.3 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。

对于本书的前两编中所讨论的架构，我们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，那么它不适宜作为此前那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[②]。“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。

---

① 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。

② 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。

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[①]。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。

[①] 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。

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”[①]。

系统总在变大[②]，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，所以架构不仅能够反映系统的“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。

[①] 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。

[②] 微之甚微，巨之愈巨，皆是系统规模的增加。

7.4 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[①]，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[②]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

[①] 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。

[②] 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。

个结果的正确性应当依赖于（其中的“+”应记为逻辑 and）：

> 1+1+1+1+…+1=1

而不应依赖于（其中“⊕”应记为逻辑 xor，“！”应记为逻辑 not）：

> 1+1+(0⊕!0)+…+1=1

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[①]。

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产生的，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

[①] 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。

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[①]，而这正是在组成论视角上无法得出的[②]。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。

7.5 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

① 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。

② 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。

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[①]：

- 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

[①] 我们此前的绝大多数讨论都是基于这两个设定的。

我们已经提到过“（架构在）时间上的可持续性”，并进而提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[①]。

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。

[①] 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。

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[①]，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。

[①] 在这样的系统中，是无法且不必讨论“佛陀拈花”这一问题的。

第 8 章 技艺、艺术与美

8.1 架构可以“学而时习”的部分

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？

8.2 死过程与活灵魂

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——技术与技艺所讨论的，总是结果的或过程的、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。

架构思想是认识系统的方法与结果[①]：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。

8.3 美

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

[①] 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。

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[①]。

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

[①] 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。

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。

8.4 架构的美

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[①]。

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[②]。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

[①] 引自《美学概论》，陈望道著。

[②] 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“知已觉”和“觉未觉”的区别。

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[①]。

就“‘美’是什么”这个问题来说，是一个哲学命题。哲学是“我见”[②]，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，在上面的讨论中，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。

---

① 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。

② 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。

8.5 舞者

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。

附一：做人、做事，做架构师——架构师能力模型解析

> 节选自《程序员》杂志 2008 年第 4 期同名文章。

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。[①]

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

[①] 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。

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，其实也是折衷的过程。构架师只有眼中见大，才知道哪些折衷可以做，而哪些不能。所谓设计评估（模型图[①]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”时可以做哪些折衷。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] 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。

附二：谈企业软件架构设计

> 节选自 ZDNET 网站 2007 年 3 月对本书作者的专访。

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。

附三：超越软件架构——组织与架构

> 本附录是《大道至易》第 1 版原书的总论。如下小段文字，聊作引文而已。
>
> 周爱民 / 2017.04.02

前些日子，有朋友说他对我的评价是“一个出色的产品经理”。这确实是远远出乎我的预料的。于是我仔细地审度了一下我的产品观念，尤其是我在产品的需求、设计与产品特性控制方面的思想与原则。我发现这些观念、思想与原则其实与我对架构的理念同出一脉。亦即是说，我的产品观与我的架构观是一致的，是同一思想的不同侧面。

有一本名为《超越软件架构》[①]的书在我的架构经历中构成了相当重要的影响。我多年来一直追问“架构是什么”，并进而将自己的架

[①] *Beyond Software Architecture: Creating and Sustaining Winning Solutions*，作者 Luke Hohmann。中文译本由中国电力出版社出版发行。

构领域从技术架构、平台架构扩展到业务架构，再后来涉足产品架构、组织架构等各个方面，与这本书给我的启发不无相关。而本附录中所谈的，便是超越了软件架构的、对《大道至简》中提及到的一个最终假设[①]：

> “工程不是做的，而是组织的”

的最终求解。它是我在多年架构实践之后，从架构领域来观察工程的所见所思和所得。作为读者，可以从中观察一个完整的“从问题出发来构建概念和基于概念的逻辑”，并最终得到“一个组织模型”的过程。

参照本书之“引言：架构师的思维”，这个过程亦即是我作为一个架构师的思维模式的全像。

[①] 出自《大道至简：软件工程实践者的思想》第 5 章。

1 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。

1.1 你在哪里？你是谁？在做什么？

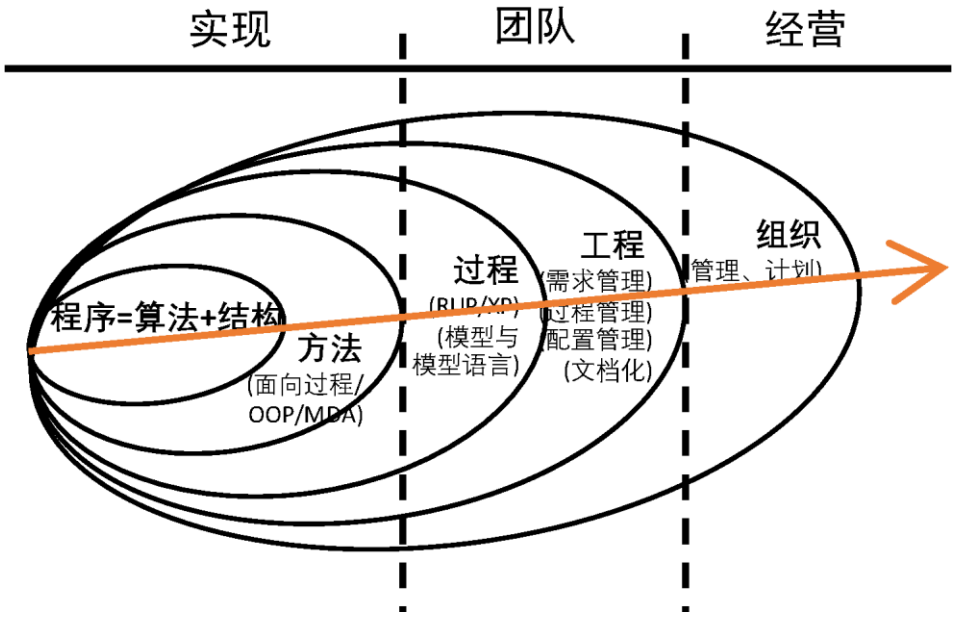
作为一个系统的组成部分，如果要观察这个系统，那么就必须清楚三个问题：你在哪里？是谁？在做什么？从这个系统中别出了“我”，才能分出“其他”，才能分清“我与其他”之于系统间的种种不同影响，从而把这个系统分析透彻。大多数人在做这样的分析的时候，忘记了观察者是“观察—被观察”系统中的一个组成部分，忘掉了“我”的位置，因而少了一半的观察。

《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。如图附 3-1 所示。

图附 3-1 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。

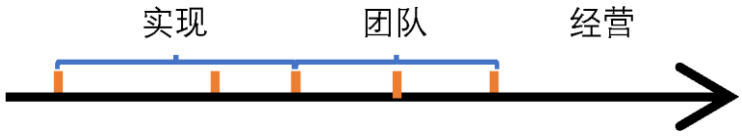
其实原因就在于，EHM 对问题少掉了一半的观察。

1.2 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。

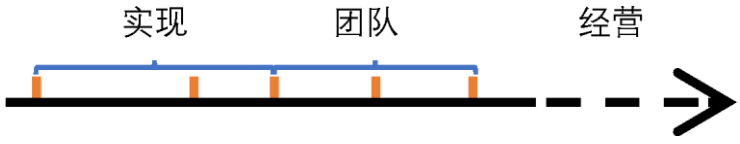
如此前所讨论的，我们可以将 EHM 模型简单地抽象为一个单向的轴线（对这里要讨论的话题来说，轴上的刻度是没有数值意义的），如图附 3-2 所示。

图附 3-2 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，如图附 3-3 所示。

图附 3-3 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。

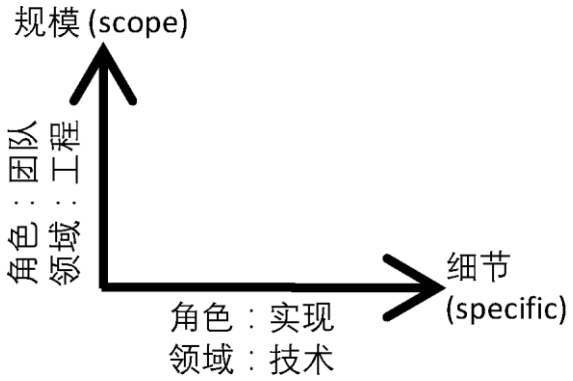
由此，《大道至简》中的 EHM 工程模型通过一个界面（临界点）来分隔二者，认为最好的法子是，让实现角色根本不知道所谓的工程，而又在遵循工程的原则下来推进项目。这种分析和推论有一定的合理性，但又忽略了其背后的本质原因，这一点在后面的讨论中将越来越明显。

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，可以描述成图附 3-4（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）。

图附 3-4 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“工程”的内在关系和冲突。

1.3 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”

[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。

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：

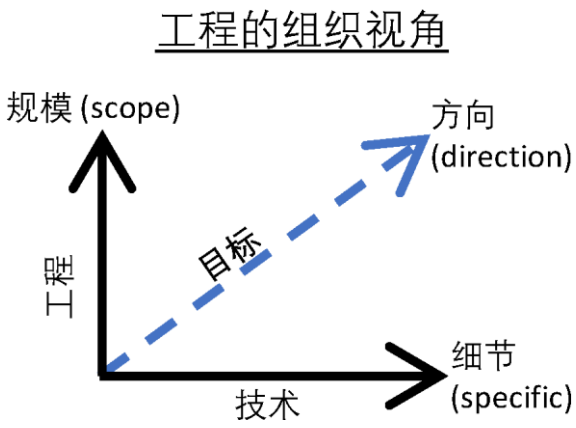
- 确定和分解了经营需求并细化成各个目标（例如发起某个项目或计划）；
- 构建了特定的组织（例如部门）来推进它。

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，我们得到图附 3-5。

图附 3-5 模型 2：“工程的组织视角”对模型 1 的影响

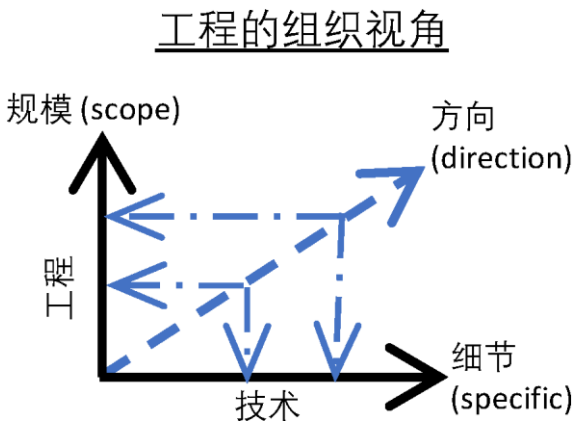


所以，我们看到了“经营”这个领域对工程的影响既深刻又浅末，既是必要条件又难以形成约束。这其中不仅仅有经营角色本身的精力问题，也涉及能力问题。因为，要确保图附 3-5 所展示的“工程活动”的实效性，那么我们还需要注意到一个事实：

> 所谓“规模”与“细节”，其实只是“目标”在两个领域中的投影。

如图附 3-6 所示。

> 图附 3-6 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[①]）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。

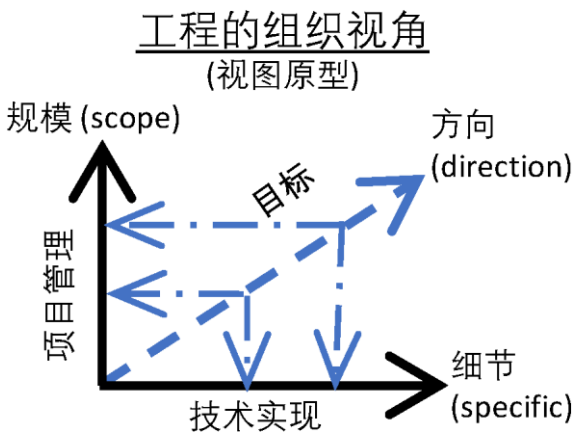
上面的第 2 条和第 3 条推论其实和我们现实的观察与实践是一致的。反过来说，也可以认为该模型体现了工程的现实状况。

1.4 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 的概念以得到图附 3-7 所示的一个原型。

图附 3-7 模型 4：对模型 3 的概念修正

[①] 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。



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[①]。

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

[①] 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，“【附三】2.5 调适：变化中的 VEO 模型”）。

当”是同一的。在现实的工程中，我们通常称之为**“产品”**[①]。

1.5 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，

经营者便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。

[①] 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。

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。

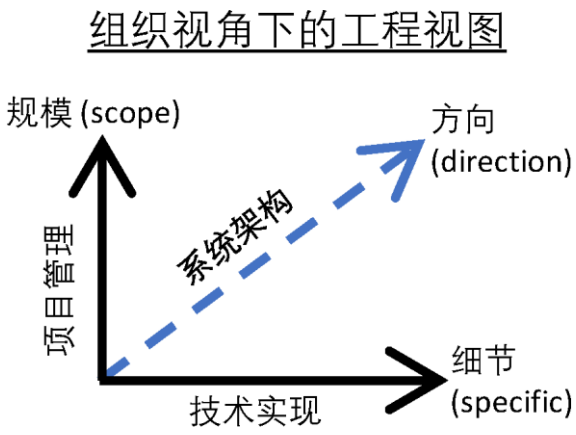
经营者选择组织工具而非其他，是有一定合理性的。首先，模型 4 在“方向”轴上的缺失是局部问题而非全局问题，因此通过组织工具来调适不会带来全局性的风险。其次，经营者可能并不期望颠覆正在进行中的阶段性目标，因此选择组织工具而非战术手段（例如裁撤项目）相对会更为温和。

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，如图附 3-8 所示。

图附 3-8 模型 5：组织视角下的工程视图



在从 EHM 模型推进到图附 3-8 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。

2 基于组织视角的观察

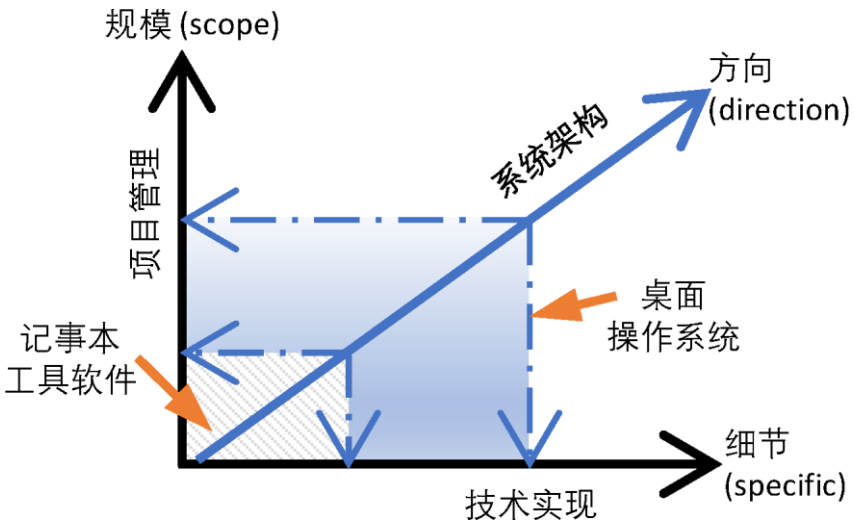
我曾经说“记事本[①]这样的软件不需要架构”，这句话并不对，因为彼时我不能将记事本作为一个系统来看待。事实上“系统”这个词并没有“大小”的限定条件，就如同说“人”这个概念本身没有“群体”的限定条件一样：并不是说一群人是人，一个人就不是人。从组织的视角来看一个系统各个独立的部分，这些部分都可称为角色。

2.1 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，图附 3-9 中规模的绝对大小是没有数值意义的）。

> 图附 3-9 模型 6：不同规模的系统在 VEO 模型上的映像

[①] 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。



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[①]。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：

[①] 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。

- 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。

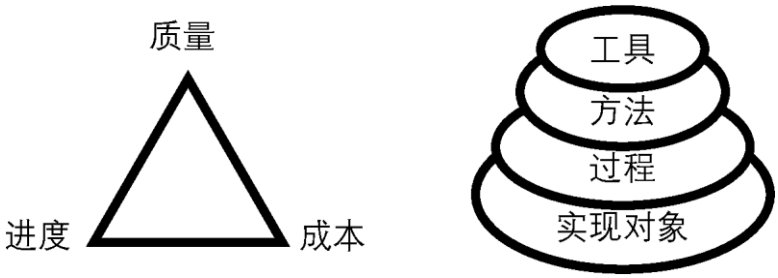
而在这个目标之下，规模问题是**项目管理**这个角色所主要关注的，更确切地说：

> 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，如图附 3-10 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。

> 图附 3-10 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。

而对于**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[①]。当把上述决策过程放在具体的项目中时，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。

[①] 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。

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[①]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（不过架构角色的分类问题并不是这里要讨论的话题）。

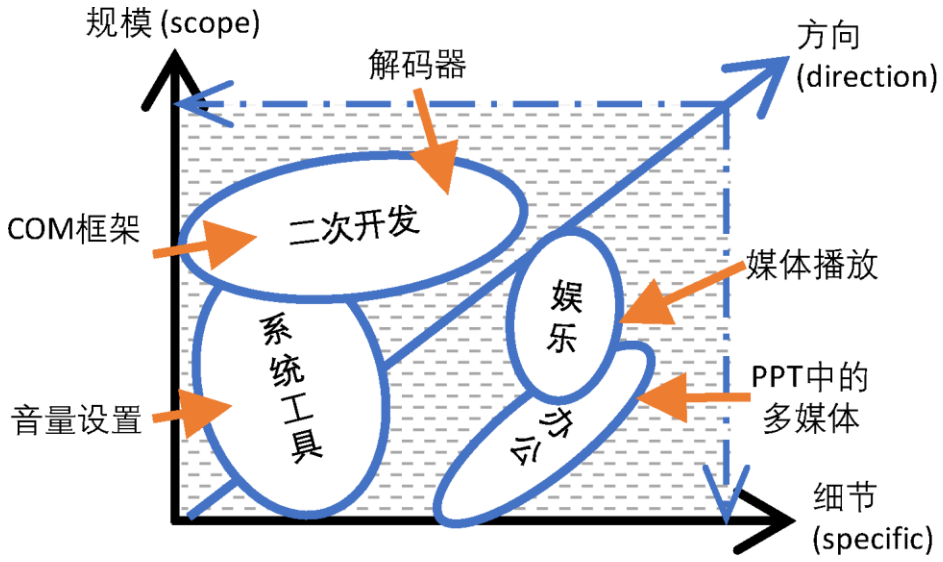
2.2 透视：一体的两面与多面

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

[①] 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。

或基本同向的，也有可能是异向的，或者无关的。图附 3-11 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。

图附 3-11 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；

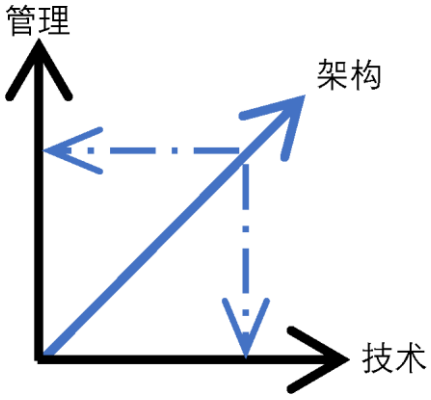
(5) 不同的方向间产生的内耗极大地增加了系统的代价，而规模与细节的失控只是这个问题在两个轴向上的表现。

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。

2.3 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附 3-12，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。

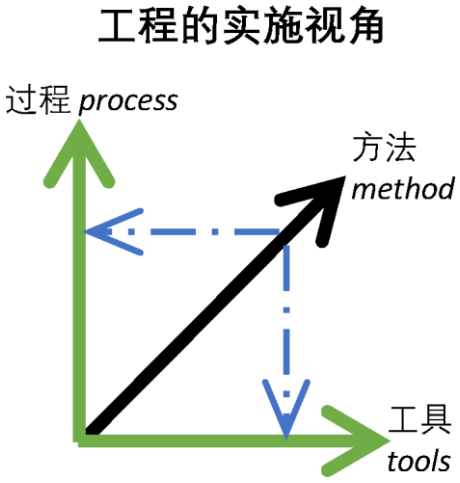
图附 3-12 技术、管理之于架构是一体两面的关系



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。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，如图附 3-13 所示[①]。

图附 3-13 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。

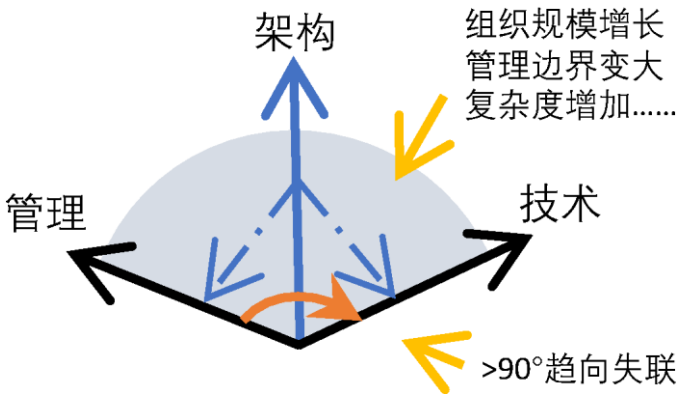
现在，为了保障更大规模下的目标，我们引入了架构角色。架构这一角色，在组织关系上介于管理与技术二者之间，在工作内容上与二者有所重叠，在职责权力上又难以对当前项目与产品产生直接效

① 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。

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，图附 3-14 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）。

图附 3-14 模型 8：组织结构调整带来的问题

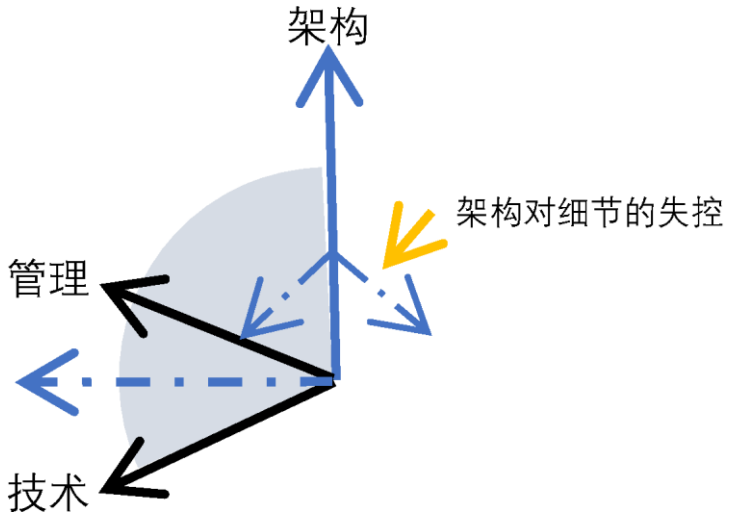


所谓组织失联，是指在组织结点之间（例如管理与技术）所工作的方向不一致、沟通成本增加，以及信息不对称——例如架构角色所展示的“目标映射”在其他角色有不同理解。进一步的结果，随着失联愈趋明显，最终有效的工作集变得越来越小。与此同时，组织的规模却在增长，管理边界与成本也增大了。这一过程，其实就是所谓的组织结构调整所带来的内耗。

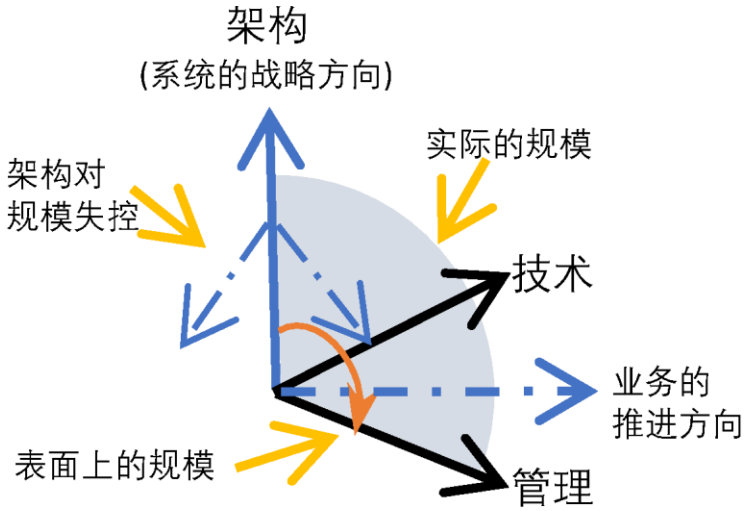
积极的管理或技术角色会趋向于弥补这种内耗，无论是二者谁占据主动（话语权或说服力），其结果无非是图附 3-15 所示的两种情况中的一种。尽管这两种模型从本质上来说是一样的——因为“实际的规模”中的内耗与模型 8 中是等量的，但是架构失控偏向于某一侧，其结果却可能不同。

图附 3-15 模型 9：积极的组织适应带来的效果

模型 9a：架构对细节的失控



模型 9b：架构对规模的失控



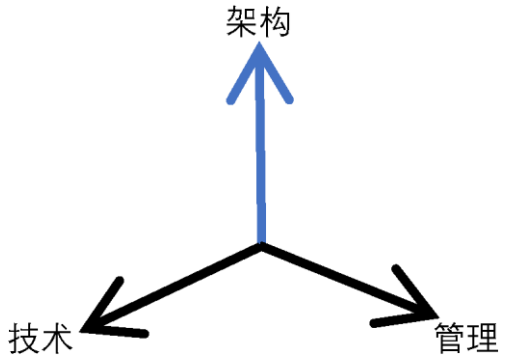
在模型 9a 中，由于架构角色对细节失控，因此他只能尽量争取管理角色的支持。如果他有足够的组织责权，管理角色又能很好地配合，那么业务会朝向架构目标推进。但即使如此，由于技术角色所能感

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，由于架构角色对规模失控，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，如图附 3-16 所示。

图附 3-16 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。

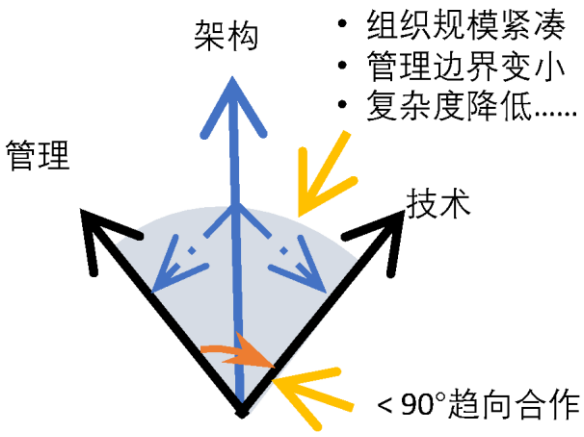
所以，如果仅仅构建（或自然形成）VEO 模型中所述的架构、管理与技术角色关系，并依赖他们完全自发地调适，模型 10 将会是最有可能的结果，也将是消耗最大的一种衡态：系统的规模最大而有效的工作集最小，并且在目标和实施上存在不对等和不确定性。

2.4 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 图上可以表现为图附 3-17。

图附 3-17 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，如图附 3-18 所示。

图附 3-18 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。

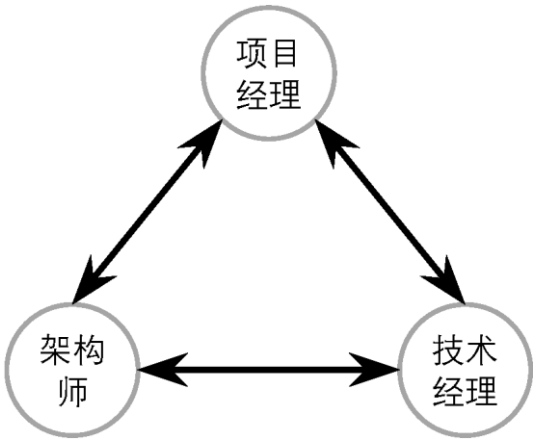
2.5 调适：变化中的 VEO 模型

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，图附 3-19 表明这三类领导角色在组织上是趋于衡态的[①]（领导力下的衡态，以下简称 LoE，即 Leadership of Equilibrium）。

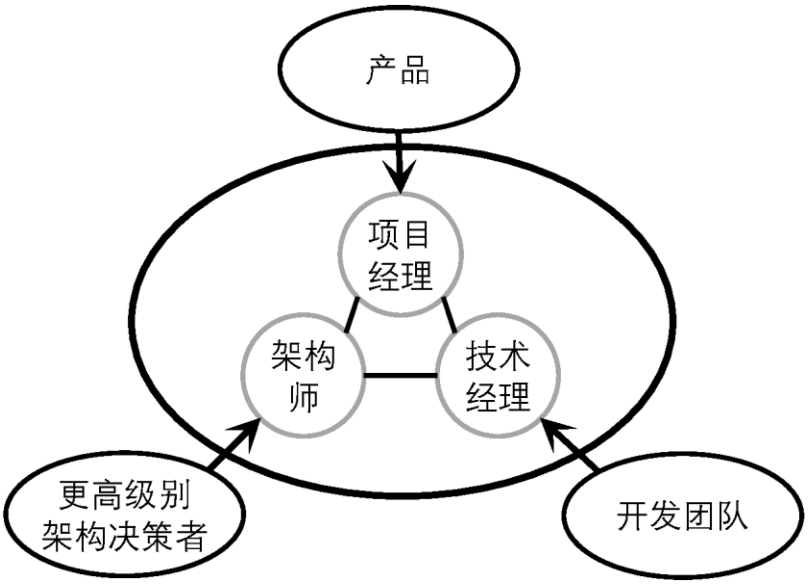
图附 3-19 （与领域相关的）领导力下的衡态



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），如图附 3-20 所示。

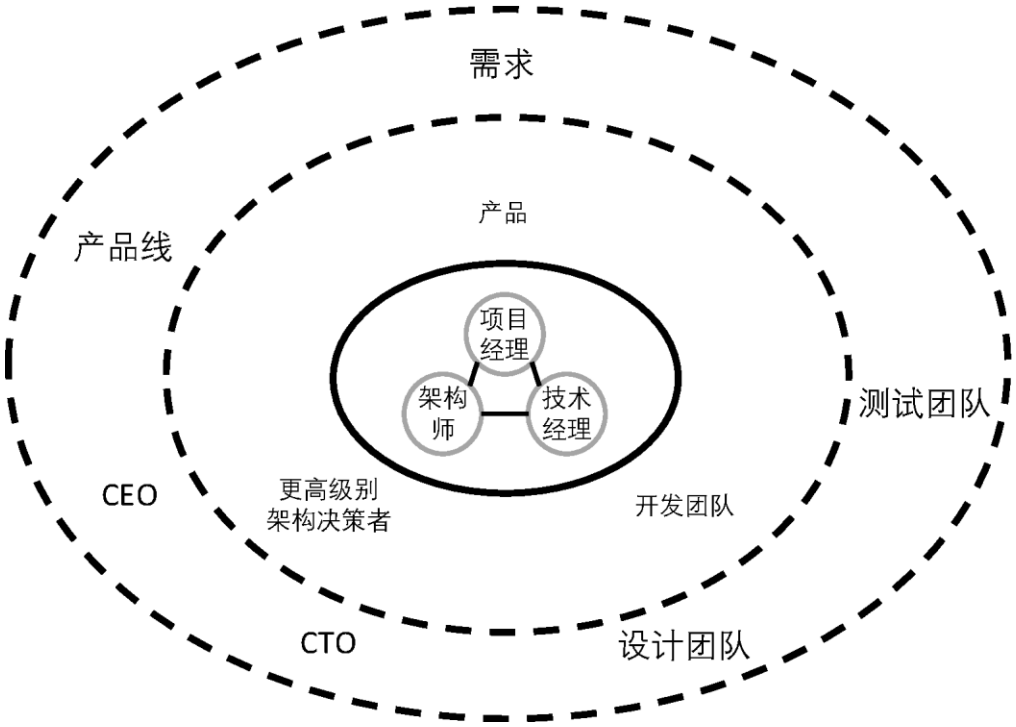
图附 3-20 外层角色的影响

[①] 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。



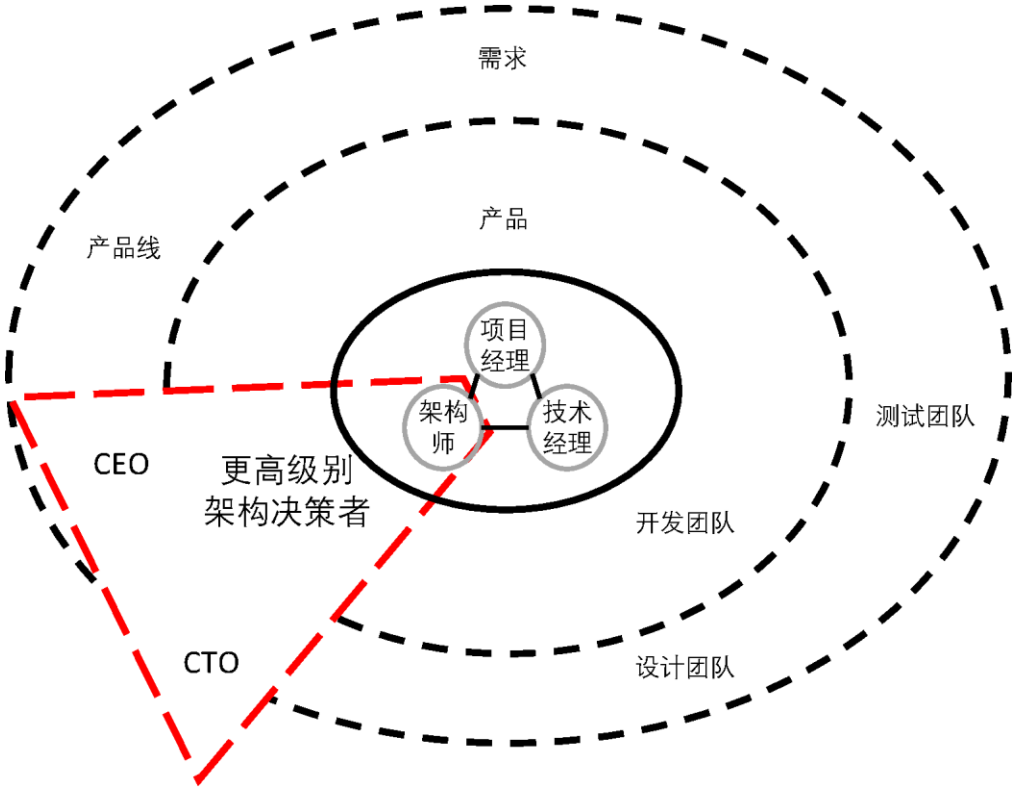
许多情况下，这些外部力量左右或影响了具体角色的实施（并不一定是决策）。这个问题随着领导责权属性的不断增加，例如可能出现的跨角色授权或领导小团体的形成，将不断扩大到整个的、全局的系统，如图附 3-21 所示。

图附 3-21 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，如图附 3-22 所示。

图附 3-22 通过合并来打破阶层（示例 1）：决策者的思想



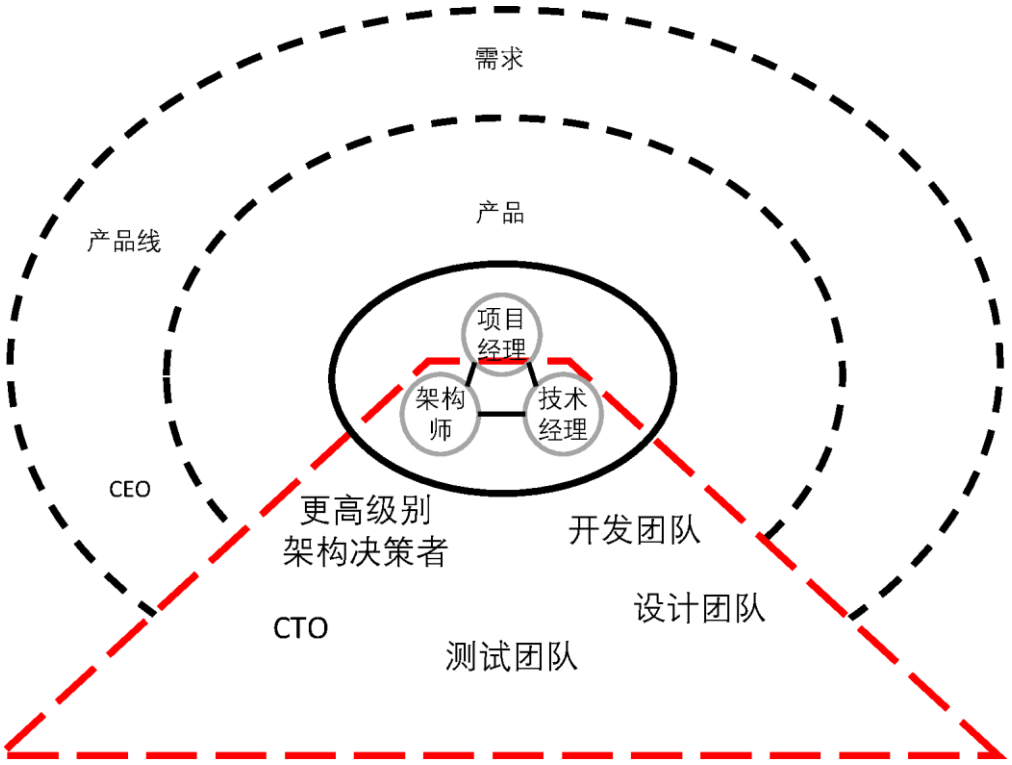
这样的组织形态一方面可能因为在相应方向上的领导缺位导致，另一方面，也是因为组织中既已形成阶层结构——并由此带来权责分配，因此“自然地转变为（而未必适宜）”由这些角色通过他们的领导力来影响项目。甚至有些时候，你看到 CEO 都被合并到这个角色中来。由于这种合并，项目常常变得非常难于控制，因为角色对项目的影响会因为“管理层次”，而非“项目自身需求”而变化。

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

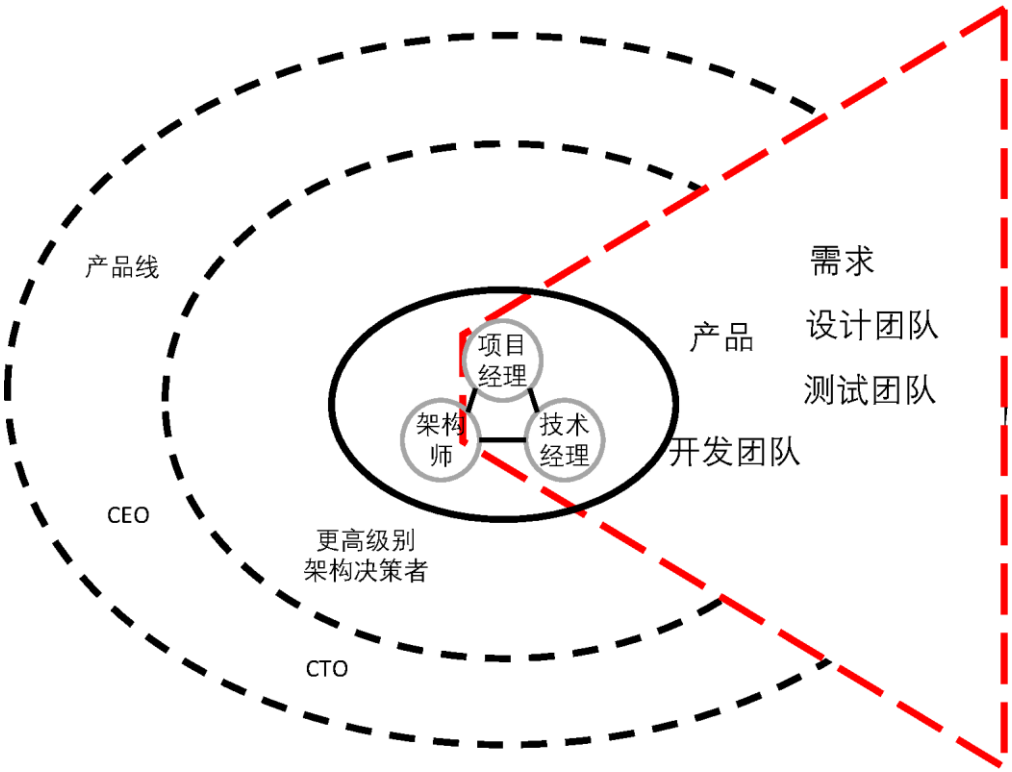
变；最后，它事实上也提供了领导角色短期实践的空间。然而更大范围的合并，通常责权更为集中，而角色与其职能也就更难区别，而且一些时候也不可避免地出现“为了形成对立”而结成的联盟。

图附 3-23 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。图附 3-24 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。

图附 3-23 通过合并来打破阶层（示例 2）：实现者的思想

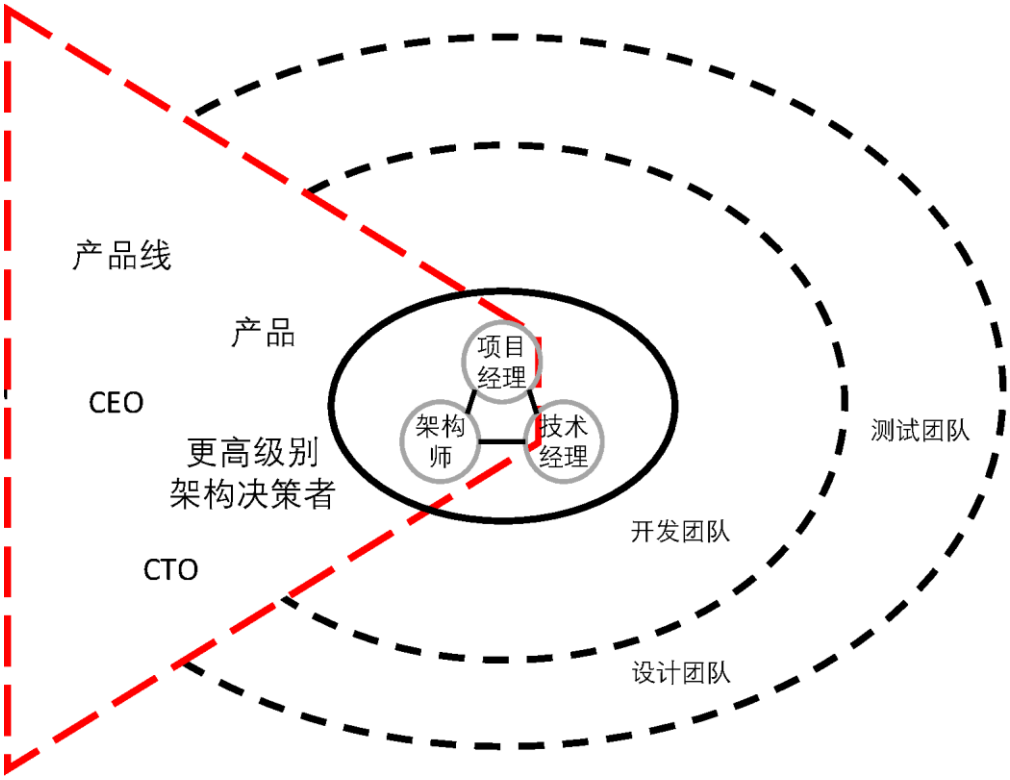


图附 3-24 通过合并来打破阶层（示例 3）：尝试者的思想



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，如图附 3-25 所示。

图附 3-25 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[①]。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[②]，如图附 3-26 所示。其中：

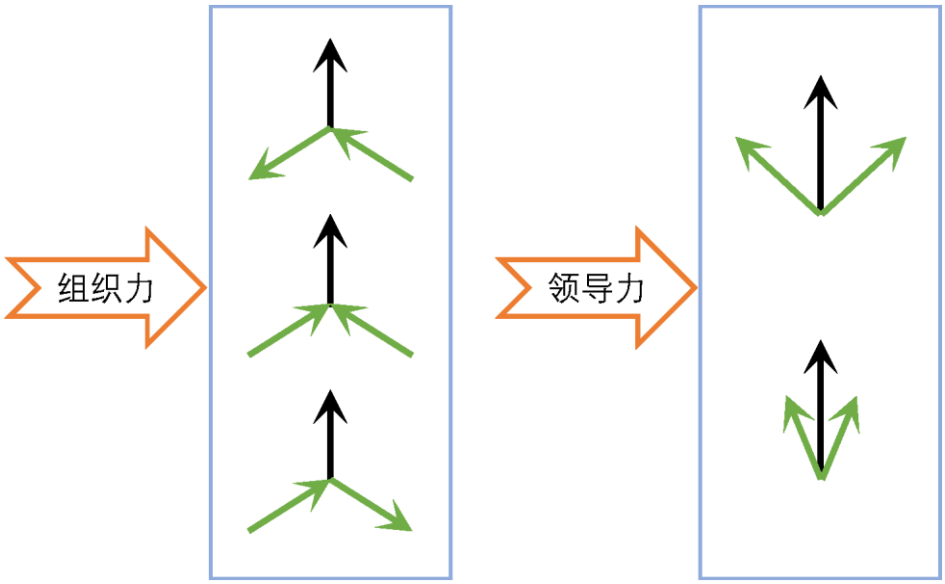
[①] 一定程度上，可以参考奥尔森有关利益与利益联盟的形成的理论。

[②] 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。

(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。

图附 3-26 组织力与领导力对组织形态的不同影响



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。