Broadview®
www.broadview.com.cn 博文视点

一线架构师实践指南

温昱 著

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

20位专家一致推荐

一线架构师实践指南

Broadview®
技术凝聚实力·专业创新出版

很值得有志成为“一线架构师”的人士学习和借鉴。

——左春 中科软总裁 中科院软件所研究员

两年来，我和我的团队应用了温老师的一些方法来开发电信行业无线网优平台这个大型平台软件，目前已经取得初步成功。

——杜海亮 天元网络公司 副总工程师

本书是从实践中来的，自然可以很好地运用到实践中去，具有很高的实践指南价值。

——宋兴烈 起步科技 总工程师

书中的三阶段理论、结构化需求与约束分析等不少概念一经指出，让人有茅塞顿开之感。书中有很实用的操作技巧，值得每一个架构师反复学习和操练，领会之后定会让您的架构设计更上一层楼。

——董振江 中兴通讯业务研究院 副院长

作者简介：

温昱　资深咨询顾问，CSAI特聘高级顾问，软件架构专家。软件架构思想的传播者和积极推动者，中国软件技术大会杰出贡献专家。十年系统规划、架构设计和研发管理经验，在金融、航空、多媒体、电信、中间件平台等领域负责和参与多个大型系统的规划、设计、开发与管理。作为资深咨询顾问，已为众多知名企业提供了卓有成效的架构培训与咨询服务。

图书分类　软件工程

ISBN 978-7-121-09540-5

9 787121 095405 >

定价：35.00元

策划编辑：徐定翔
责任编辑：徐定翔
责任美编：杨小勤

一线架构师实践指南

温昱 著

电子工业出版社
Publishing House of Electronics Industry
北京 · BEIJING

内 容 简 介

本书致力于为一线架构师，以及软件企业提供务实有效的架构设计方法指导。

本书从架构师经常遇到的困惑出发，总结软件架构设计中经常遇到的问题，提出“方法体系必然是软件业界未来发展的重大趋势”这一观点；之后，详细阐述了软件架构设计三个阶段（Pre-Architecture 阶段、 Conceptual Architecture 阶段和 Refined Architecture 阶段）中的各个具体环节，并给出了最佳的实践原则和方法，内容涵盖“需求进，架构出”的整个过程。

图书在版编目（CIP）数据

一线架构师实践指南 / 温昱著.—北京：电子工业出版社，2009.10

ISBN 978-7-121-09540-5

I.…　II.温…　III.软件设计－指南　IV.TP311.5-62

中国版本图书馆 CIP 数据核字（2009）第 166549 号

策划编辑：徐定翔

责任编辑：徐定翔

印　　刷：北京智力达印刷有限公司

装　　订：北京中新伟业印刷有限公司

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱　邮编 100036

开　　本：787×1092　1/16　印张：13.5　字数：300 千字

印　　次：2009 年 10 月第 1 次印刷

印　　数：4 000 册　　定价：35.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

前辈推荐

（以姓氏笔划为序）

本书系统介绍了当前软件架构设计领域先进的 ADMEMS 方法体系，并以作者十余年来在软件开发实践和研究中积累的丰富经验，在论述架构设计不同阶段的分析方法与设计技术的同时，给出了相应的实践策略、实践套路及有用的设计案例。本书具有极强的实用性，不但是一线架构师及希望成为软件架构师者的福音，对我国软件业界在软件架构相关方面的研究工作也有一定的推动作用。值得一提的是，本书文笔生动，深入浅出，议论充满睿智，读来常令人有如沐春风之感，在技术类书籍中也是不可多得的。

——杨晋兴　中航集团公司 631 研究所研究员　前系统软件室主任

作者在本书中提出了 ADMEMS 架构设计方法学，特别是详细论述了逻辑架构设计的 10 条经验，以及基于鲁棒图进行初步设计的 10 条经验。这些经验既是作者亲身的实践总结，又概括了业界的有效实践。作者还运用贯穿案例、大型网站案例等形式，将实践经验与原理整合起来，以帮助读者理解和掌握 ADMEMS 架构设计方法学的精髓。本书不仅生动地反映出作者的创造性思维和对学术的刻苦耕耘，又反映出作者对架构学的崇高历史责任感。我相信本书的出版，不仅对架构师们有很好的参考价值，而且对推动架构学界的深入研究具有重要意义。

——周伯生　北航计算机学院教授　博士生导师　美国 SDPS 学会院士

编写一套软件系统并不困难，但要编写一套优质、高效的软件系统却是极大的挑战。一套优质的软件需合理设计，功能需求、未来应用环境、硬体组合、数据处理要求、计算逻辑、用户分布、使用习惯等多方面因素都是系统架构师在软件工程的设计阶段要思考及解决的问题。软件工程的架构师犹如建造工程的建筑师一样，一些建筑师能够最终成为“大师”，主要是他们的建筑设计除了能够满足应用需求外，还能结合周边环境，拥有独特的组合理念和创意。把握软件的架构设计技巧和方法，才能够带出软件创新的成果。《一线架构师实践指南》提供从业人员这方面的信息，透过简易的说明和分析，让读者理解如何才能够客观地为客户设计高效和优质的计算机软件，是成为真正软件工程师的第一步，是未来软件大师的实践指南。

——黄结良　南开大学软件学院　教授

专家推荐

（以姓氏笔划为序）

架构师不仅仅是名片上的一个头衔，他本人必须在熟悉客户须要做什么的基础上，伏下身子带领各个开发团队攻克难题并优化系统组成间的关联。架构师不应是单纯的界面设计人员——只能出产幻灯片和插图；他应能用系统而缜密的步骤帮助团队完成更“好”的产出。很喜欢读温先生对于架构设计的著作，对本书“一线”的概念更加推崇，毕竟没有“一线”就直接去尝试所谓的“宏观”、“超大”似乎不够稳妥。

——王 翔 全国海关信息中心 高级技术架构师

什么是软件架构？目前似乎没有标准的答案，它的本质是给一个软件系统做一个蓝图式的表述。面对这类表述，我们可以有多种方式进行概念抽象和细节忽视，就像给现实世界作画（也是一种蓝图方法），可以使用各种方法，如：写实，素描……。我们还可以形成不同风格，如：抽象派、印象派……。温昱先生在介绍了“方法体系”的基础上，给出了自己关于软件架构的方法和看法，很值得有志成为“一线架构师”的人士学习和借鉴。

——左 春 中科软科技股份有限公司总裁 中科院软件所研究员

架构是什么，每个人都可以说上两句，但是很少有人能说得清楚。如何做架构呢？大家都知道其无比重要，但是大部分人还是一头雾水、不知所措。怎么讲架构呢？晦涩难懂，两分钟就能让人进入梦乡的比比皆是，把架构讲得像故事，读起来让人感觉津津有味、流连忘返的书真是世间少有。

《一线架构师实践指南》正是这样一本书，对于那些有志于成为架构师的人来说，它既是思想的启蒙者，又是行动的指导者，让你在不知不觉中学习、成长。如果还认为架构是那么高深莫测、遥不可及，你不妨看看它。

——齐书阳 《软件世界》杂志社 主编

初识温昱老师是在 2007 年一场由他主讲的“软件架构师”的培训会上，他对软件架构的独到理解及实际经验形成的贯穿案例给我留下了深刻的印象。

两年来，我和我的团队应用了温老师的一些方法来开发电信行业无线网优平台这个大型平台软件，目前已经取得初步成功。在这个平台中，概念性架构设计、5 视图法细化架构设计等方法都在团队中达成统一认识，并较好地应用在实际工作中，特别是非功能需求设计的方法在用户话单分析和无线测量报告等海量数据处理方面的实践取得了明显的成效。

《一线架构师实践指南》一书秉承作者在架构设计上注重方法、注重实践的一贯思想，围绕“需求进、架构出”全过程实践指导的方法体系，对软件架构设计的三个阶段一个贯穿环节进行了详尽的描述，其中无处不在的案例对 ADMEMS 体系做了生动的描述和分析，确实是一本在架构设计领域具有实践指导意义的、难得的好书。

每天忙碌于软件设计的和程序开发的软件业朋友们，不妨停下脚步来用心拜读一下这本书，它会从多个视角告诉你在不同阶段应该做什么样的事情，相信大家会受益匪浅。

——杜海亮 天元网络公司 副总工程师

作为一名软件工程和架构设计的实践者，我追求构建“可靠、适用和易扩展”的系统，应用适当的技术、工具和方法来解决实际项目中的需求和问题。本书提供了丰富而实用的理论和技术实践策略，既适合初学者学习也适合经验丰富的软件工作者深入体会，具有很高的参考价值。

——李胜利 东方电子资深架构师 高级项目经理

一口气将《一线架构师实践指南》专家评荐版阅读完，感觉意犹未尽。针对软件架构设计相对“神秘”、“高深莫测”来讲，急需方法论使之有章可循。这本书正是在架构设计的方法论方面、设计细节量化方面、设计应采取的原则方面都做了针对性总结和概括，具有重大的实践指导意义和推广价值，为一线架构师不可多得的理论指导书！

——李哲沫 东软集团电信事业部研发二部部长
资深咨询顾问 东北大学客座讲师

《一线架构师实践指南》一书，深入浅出，对中大型系统的架构设计起到了航标灯的作用，不仅解决了资深架构师的困惑，而且对新手具有重大的指导意义。它把抽象的理论落实到实际的可操作的范围，令人折服。

——宋英 西门子公司 资深 IT 专家

软件架构设计很难，难在只能意会，很难言传。

本书作者不但具有丰富的架构设计方面的理论知识，而且有多年的架构设计和咨询实践经验。基于这些理论知识和实践经验，作者形成了关于架构设计方面的核心主张，并且提出了非常具有指导和实践意义的方法体系——ADMEMS。细细体会这些核心主张和 ADMEMS 方法，发现似曾相识，特别有共鸣。原来我们在平时的架构设计中，竟不知不觉地在使用这些主张和方法，但是没有总结出来。本书作者的高明，在于系统地总结和抽象，在于言传。

我非常愿意向读者推荐此书，因为本书是从实践中来的，自然可以很好地运用到实践中去，具有很高的实践指南价值。

——宋兴烈 起步科技 总工程师

在软件行业，“架构”是一个很时髦的词汇，架构师是很多年轻人梦寐以求的“金领”职位。遗憾的是，对于如何培养出优秀的架构师，特别是如何指导架构师进行设计实践的书籍很少。8 年前，我就开始从事软件架构领域的研究与实践，阅读了不少文献，但直到读了温昱先生的这本书后，才悟出“架构实践”的内涵，才真正知道该如何实现“需求进，架构出”的过程。 该书对于一个架构设计老兵尚且有此帮助，更何况新手呢?

——张友生 博士 希赛网首席架构师 希赛教育首席专家

经常同温昱老师深入交流架构设计对产品和系统质量保证的影响，温老师在架构设计的理论与实践方面有深厚的功底和丰富的经验，他多年磨练的解决实际问题的精湛能力给企业和用户带来了很大的收益。《一线架构师实践指南》的出版，是架构设计实践领域的突破，相信书中丰富实用的案例、深入浅出的理论、清晰流畅的表达，以及耐人寻味的故事会让读者回味无穷。

——陈漫薄 中国软件评测中心 技术总监

本书是温昱先生继《软件架构设计》之后的又一力作，实属原创中文软件架构图书中的奇葩。这两部姊妹篇，将作者多年来在软件架构设计方面的实践经验与独到见解，用中国程序员能接受的讲解方式，逐一展现给读者。《一线架构师实践指南》所讲述的方法原理和实践经验，对指导架构设计实践具有非常实用的参考价值。

——罗景文 IBM developerWorks 中国网站

架构是科学，也是艺术，本书化架构的艺术为科学，让架构变成可被传承可被学习的科学。无论你已经是架构师，还是想成为架构师，你都应该将这本书摆在床头，经常翻阅，并且按照书上的方法指南实践，直到将架构方法烂熟于胸，这样你也能像温昱那样成为一个优秀的架构师，设计架构的时候得心应手。

——周愚 IBM 高级架构师

温昱先生是我过去的同事，他的协助和在项目中对架构炉火纯青的运用使我受益匪浅。我向来认为“架构师”分量很重，并非在个别项目中运用了一些软件架构的思想或设计模式就能称为“架构师”。只有从实践中来，再将架构理论运用于实践，才能真正称为理解了“架构”，而温昱先生就是个中翘楚。非常喜欢读他的书，并乐于向大家推荐。

——**颜泽中 博士** 日本贝赛莱多媒体信息技术有限公司 软件部部长

看了温昱的这本《一线架构师实践指南》，我不由得想起自己经常分享的总结：“我们并不缺乏软件工程的方法，真正缺乏的是在实践中有效地组合应用它们的体系”，需求工程是这样，架构设计也是这样。而 ADMEMS 正是架构领域的指路明灯，它架构在成熟方法论这一巨人上，构建在作者多年来跨不同领域、不同平台的架构设计经验的基础上。

正如作者在书中所说的那样，架构是一门艺术；何为艺术呢？艺术是源于生活、高于生活的东西；换句话说，没有真实的生活体验就没有艺术。本书中那些让人倍感亲切的场景，毫不陌生的困惑，都使得这本书更加贴近实践，更容易让读者在实践中有效地应用本书所介绍的方法，也更加符合“艺术”的定义。

相信所有从事软件架构设计、详细设计、开发工作的从业人员都能够从本书中获得清晰的思路、可行的方法；因此我强烈推荐大家不要错过这本难得的“内功心法”。

——**徐锋** 独立咨询顾问 需求过程框架 SERU 创始人 CSAI 首席顾问

架构设计对项目至关重要，做好非常不易；相关理论和书籍不少，真正实用的则不多，温昱先生大作是个中翘楚。作者集 10 多年实践和研究，形成一套实用性强、非学院式的体系，对做好架构设计富有指导价值。书中的三阶段理论、结构化需求与约束分析等不少概念一经指出让人有茅塞顿开之感。书中有很实用的操作技巧，值得每一个架构师反复学习和操练，领会之后定会让您的架构设计更上一层楼。

——**董振江** 中兴通讯业务研究院 副院长

温昱是《程序员》杂志的作者，也是我们 SD2.0 大会邀请的讲师。几年来，CSDN 和他保持着良好的合作关系，而我在合作过程中也逐渐了解了他在架构方面的经验和积累。中国从来不缺理论家，中国软件领域尤其不缺理论家，我们缺少的是来自第一线、将实践经验提升到理论高度再反馈回实践的人。温昱就是这样一个人，他的《一线架构师实践指南》，也是这样一本书。我乐意向架构师或有志于成为架构师的读者推荐这本书。

——**韩磊** CSDN 总编辑

架构师是一种神秘的职业，成为一名合格的架构师是每个开发者的梦想。成为合格的架构师难在预见系统问题的思考方式，温昱将多年架构经历积累而成的宝贵经验传授给我们，非常难得。书中既有架构各个阶段的方法论指导，又有软件和网站架构的实战演练，是成为合格架构师必备的指南。

——曾登高 CSDN 技术总监

本书针对新老架构设计人员在实际工作中经常遇到的困惑，结合对典型案例的分析，以 ADMEMS 方法由浅入深地给出了相应的对策，实战性极强。本人认为此书实乃业界相关书籍中的一朵奇葩，强烈建议新老架构设计人员人手一本，作为将来工作中的指导参考用书。

——靳向阳 加拿大 IBM 软件工程师

序

方法之于个人，乃至软件业，都是至关重要的。对架构新手，方法是陌生之地的指路明灯，避免架构设计者不知所措（这很常见）；对架构老手，方法是使经验得以充分发挥的思维框架，指导架构设计者摆脱“害怕下一个项目”的心理和“思维毫无章法”的状态；对软件业而言，方法是整个产业“上升一个层次”的“内功”，没有“内功”为基础，单靠“外力”促进软件产业升级是不现实的。

本书致力于为一线架构师，以及软件企业提供务实有效的架构设计方法指导。

为什么这么多架构师总是抱怨需求呢？因为不少架构师不懂需求，而更多架构师缺乏需求的大局观。为此，可以看看本书【第 1 部分 Pre-architecture 阶段】的“ADMEMS 矩阵方法”、以及“约束性需求的四种类型”等内容。

设计稳定的架构，首要的一点是什么呢？是概念架构必须稳定。为此，可以看看本书【第 2 部分 Conceptual Architecture 阶段】是如何展开阐述“重大需求塑造概念架构”的。

如何更合理地将系统切分为子系统呢？答案是遵循职责分离原则、通用专用分离原则、技能分离原则、工作量均衡原则等设计思想的要求。本书【第 3 部分 Refined Architecture 阶段】讲解了分层的细化、分区的引入、机制的提取等实践技巧。

回顾过去，我在金融、航空、多媒体、电信、中间件平台等领域的职业经历中，幸运地遇到了很多良师益友，他们的智慧和无私使我受益匪浅；近几年，在软件企业一线开展架构培训与咨询工作时，认真务实的客户让我进一步开阔了视野，了解了软件业一线的现状……这些，都是本书所讲述的架构设计方法体系形成和发展的原动力。所以，由衷感谢：所有帮助和支持过我的前辈、专家、客户！

可通过 shanghaiwenyu@163.com 与我联系，欢迎探讨、批评、指正。

资深咨询顾问　温昱

2009 年 8 月于上海

Pre-Architecture阶段

Pre-architecture的故事
Pre-architecture总论
需求结构化与分析约束影响
确定关键质量与关键功能

Conceptual Architecture阶段

概念架构的故事
Conceptual Architecture总论
初步设计
高层分割
考虑非功能需求

Refined Architecture阶段

细化架构的故事
Refined Architecture总论
逻辑架构
物理架构、运行架构、开发架构
数据架构的难点：数据分布

专题：非功能目标的方法论

故事：困扰已久的非功能问题
总论：非功能目标的设计环节
方法："目标－场景－决策"表

联系博文视点

您可以通过如下方式与本书的出版方取得联系。

读者信箱：*reader@broadview.com.cn*

投稿信箱：*bvtougao@gmail.com*

北京博文视点资讯有限公司（武汉分部）

湖北省 武汉市 洪山区 吴家湾 邮科院路特 1 号 湖北信息产业科技大厦 1402 室

邮政编码：430074

电　　话：027-87690813

传　　真：027-87690595

若您希望参加博文视点的有奖读者调查，或对写作和翻译感兴趣，欢迎您访问：*http://bv.csdn.net*

关于本书的勘误、资源下载及博文视点的最新书讯，欢迎您访问博文视点官方博客：*http://blog.csdn.net/bvbook*

目 录
content

第1章
绪　论

软件架构在不断发展，但它仍然是一个尚不成熟的学科。

——Len Bass，《软件构架实践（第2版）》

推动软件工程研究不断发展的，常是实际生产或使用软件时遇到的难题（Software engineering research is often motivated by problems that arise in the production and use of real-world software）。

——Mary Shaw，《The Golden Age of Software Architecture》

架构设计能力，因掌握起来困难而显得珍贵。

本章概括一线架构师经常面对的实践困惑，并点出 ADMEMS 方法的应对之策。

1.1　一线架构师：6 个经典困惑

一线架构师经常面对的实践困惑，可以用图 1-1 来概括。其中，涉及了“4 个实际问题的困惑”，以及“两个职业发展的困惑”。

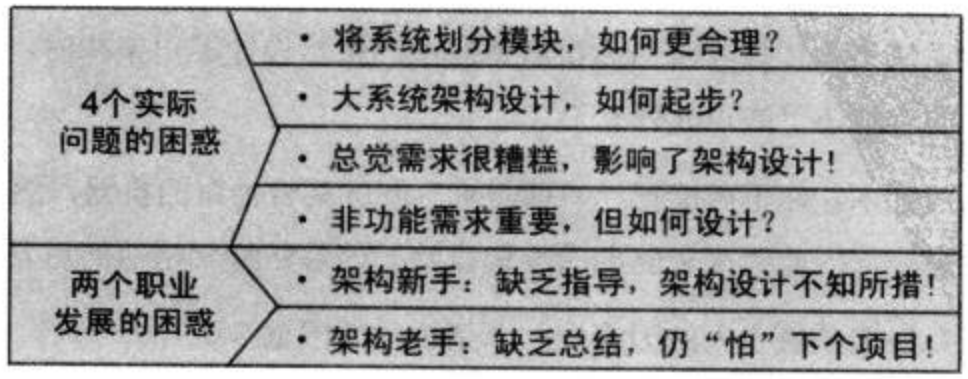


图 1-1　一线架构师的 6 个经典困惑

1.2 本书的4个核心主张

画龙须点睛。

在介绍具体方法之前，先来阐释本书的4个核心主张：

- 方法体系是大趋势。
- 质疑驱动的架构设计。
- 多阶段方法。
- 内置最佳实践的方法。

这4个核心主张可帮助读者领会ADMEMS方法之精髓。

1.2.1 方法体系是大趋势

单一方法已捉襟见肘。一线架构师真正需要的，是覆盖“需求进，架构出”全过程的实践指导——只有综合了不同方法优点的“方法体系”才堪此重任。本书认为，方法体系必然是软件业界未来发展的重大趋势之一。

本书将要系统介绍的方法体系的名字——ADMEMS，正是“Architectural Design Method has been Extended to Method System”的缩写。是的，ADMEMS方法不是“单一方法”，而是由多个各具特点的方法组成的“方法体系”。ADMEMS方法通过它的名字亮明了其核心主张。

ADMEMS方法命名由来

ADMEMS是“Architectural Design Method has been Extended to Method System（架构设计方法已经扩展到方法体系）”的缩写。

1.2.2 质疑驱动的架构设计

从根本上讲，架构设计毫无疑问是需求驱动的，而不是模型驱动的。

但需求驱动的说法不太传神——当你很清楚需求却依然设计不出架构时，就足以说明“需求驱动的架构设计”的总结还“缺点儿什么”。

架构设计是一门艺术，你不可能把“一桶需求”倒进某台神奇的机器，然后等着架构设计自动被“加工生产”完毕，因此“需求驱动的架构设计”的总结给架构师的启发不够。

缺点儿什么呢？答案是缺“人的因素”、“架构师的因素”！

本书将不断阐释架构设计实际上是个“质疑驱动的过程”：需求，被架构师的大脑（而不是自动）有节奏地引入架构设计一波接一波的思维活动中。例如，作为架构师，当你的架构设计进

行到一半时，你可以明显感觉到：这个架构设计的中间成果，还须要进一步通过“质疑”引入更多“质量属性”，以及“特殊功能场景”来驱动后续架构设计工作的开展。

在保留“需求驱动的架构设计”所有正确内涵的同时，“质疑驱动的架构设计”告诉架构师：你的头脑，才是架构设计全过程的发动机。质疑意识，是架构师最宝贵的意识之一。

至于有的专家提倡的“用例驱动的架构设计”观点，则有严重缺陷，3 句话足以揭示这一点：

- 需求 = 功能 + 质量 + 约束。
- 用例是功能需求的实际标准。
- 用例涉及、但不涵盖非功能需求。

1.2.3 多阶段还是多视图?

架构设计的多视图方法很重要，但是，架构设计方法首先应当是多阶段的，其次才是多视图的。如图 1-2 所示。

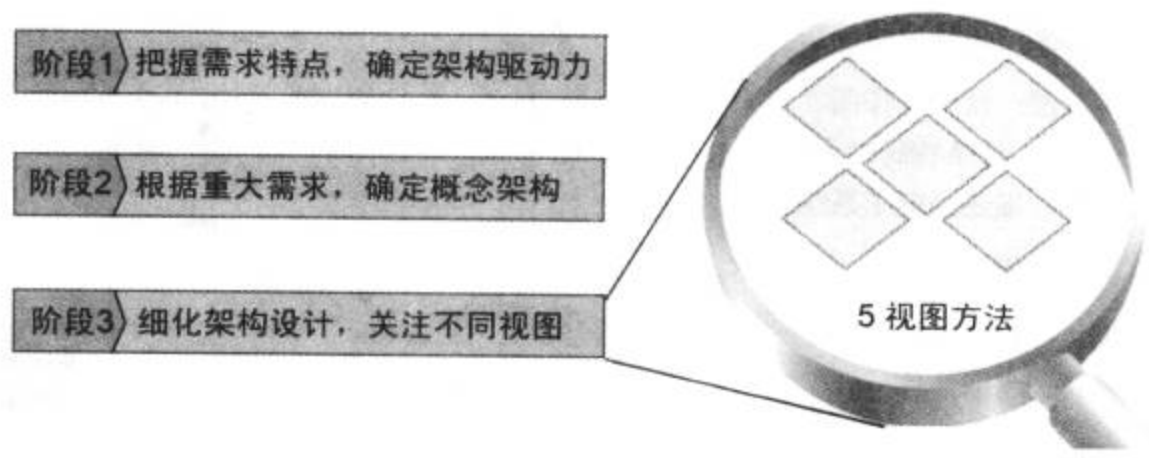


图 1-2 架构设计方法首先应当是多阶段的，其次才是多视图的

一句话，先做后做——这叫阶段（Phase），齐头并进——这叫视图（View）。

本书认为，任何好的方法（不局限于软件领域），都必须以时间为轴来组织，因为这样才最利于指导实践。

架构设计只需多视图方法，看上去很美，其实并不足够。实际上，大量一线架构师早已感觉到多视图方法的“不足够”。例如，想想投标：

- 一方面，投标时，要提供和讲解《方案建议书》，其中涉及架构的内容。
- 另一方面，团队并行开发时，需要《架构设计文档》供多方涉众使用。
- 但是，投标时讲的“架构”和并行开发时作为基础的“架构”在同一个抽象层次上吗？绝不可能。前者叫概念架构，后者叫细化架构。如果投标失败，细化架构设计根本就不须要做了。
- 结论，概念架构设计和细化架构设计，是两个架构阶段，不是两个架构视图。

1.2.4 内置最佳实践

方法不应该是个空框框，应融入最佳实践经验。相信业界很多专家都正朝着这个方向迈进。

ADMEMS 方法中融入了笔者的哪些实践经验呢？仅举几例：

- 逻辑架构设计的 10 条经验（如图 1-3 所示）。
- 质疑驱动的逻辑架构设计整体思路（如图 1-4 所示）。
- 基于鲁棒图进行初步设计的 10 条经验。
- ADMEMS 矩阵方法。
- 约束的 4 大类型。
- ……

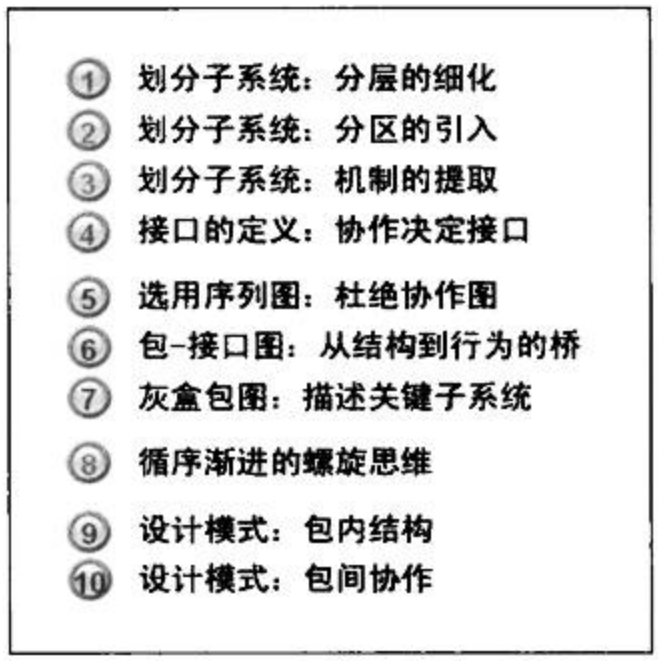


图 1-3 逻辑架构设计的 10 条经验

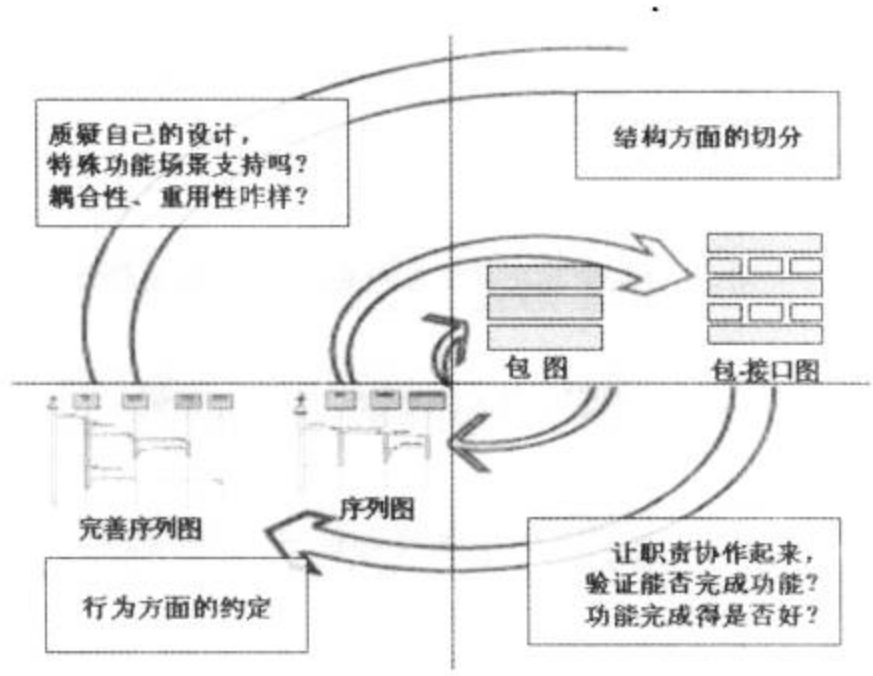


图 1-4 质疑驱动的逻辑架构设计整体思路

1.3 ADMEMS 方法体系：3 个阶段，1 个贯穿环节

作为方法体系，ADMEMS 方法通过 3 个阶段和 1 个贯穿环节，来覆盖“需求进，架构出”的架构设计完整工作内容。

图 1-5 说明了“3 个阶段”在整个方法体系中的位置。具体而言：

- 预备架构（Pre-architecture）阶段（简称 PA 阶段）。
  - 最大误区：架构师是技术人员不必懂需求。
  - 实践要点：摒弃“需求列表”方式，建立二维需求观。
  - 思维工具：ADMEMS 矩阵等。

- 概念架构（Conceptual Architecture）阶段（简称 CA 阶段）。
  - 最大误区：概念架构 = 理想设计。
  - 实践要点：重大需求塑造概念架构。
  - 思维工具：鲁棒图、目标-场景-决策表等。
- 细化架构（Refined Architecture）阶段（简称 RA 阶段）。
  - 最大误区：架构 = 模块 + 接口。
  - 实践要点：贴近实践的 5 视图法。
  - 思维工具：包图、包-接口图、灰盒包图、序列图、目标-场景-决策表等。

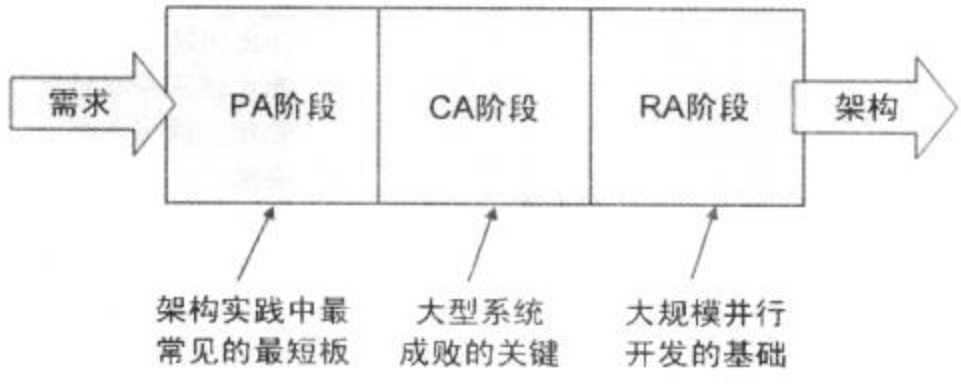


图 1-5 ADMEMS 方法体系包含 3 个阶段

值得强调的是，上述 3 个阶段之间的先后顺序有极大的实际意义，否则就不能称其为“阶段”了：

- 试想，在 Pre-architecture 阶段对需求理解不全面（例如遗漏了需求）、不深入（例如没有发现“高性能”和“可扩展”是两个存在矛盾的质量属性），后续设计怎会合理？
- 试想，Conceptual Architecture 阶段的概念架构设计成果没有反映系统的特点就“冲”去做 Refined Architecture 设计，是不是必然造成更多的设计返工？

“1 个贯穿环节”，指的是对非功能目标的考虑。

1.3.1 Pre-architecture 阶段：ADMEMS 矩阵方法

Pre-architecture 阶段的使命，可以概括为一句话：全面理解需求，从而把握需求特点，进而确定架构设计驱动力。其中，ADMEMS 矩阵居于方法的核心。

“ADMEMS 矩阵”又称为“需求层次-需求方面矩阵”（如表 1-1 所示），帮助架构师告别需求列表的陈旧方式，顺利过渡到二维需求观，借此避免遗漏需求、并进一步理清需求间关系和发现衍生需求。

表 1-1 ADMEMS 矩阵的基础是二维需求观

	功能	质量	约束
业务级需求	业务目标	快、好、省	技术性约束 法规性约束 技术趋势 竞争因素与竞争对手 遗留系统集成 标准性约束 分批实施
用户级需求	用户需求	运行期质量	用户群特点 用户水平 多国语言
开发级需求	行为需求	开发期质量	开发团队技术水平 开发团队分布情况 管理：保密要求 安装 开发团队磨合程度 开发团队业务知识 管理：产品规划 维护

1.3.2 Conceptual Architecture 阶段：重大需求塑造做概念架构

概念架构 ≠ 理想化架构。所以，必须考虑包括功能、质量、约束在内的所有方面的需求。图 1-6 说明了 ADMEMS 方法推荐的概念架构设计的高层步骤。

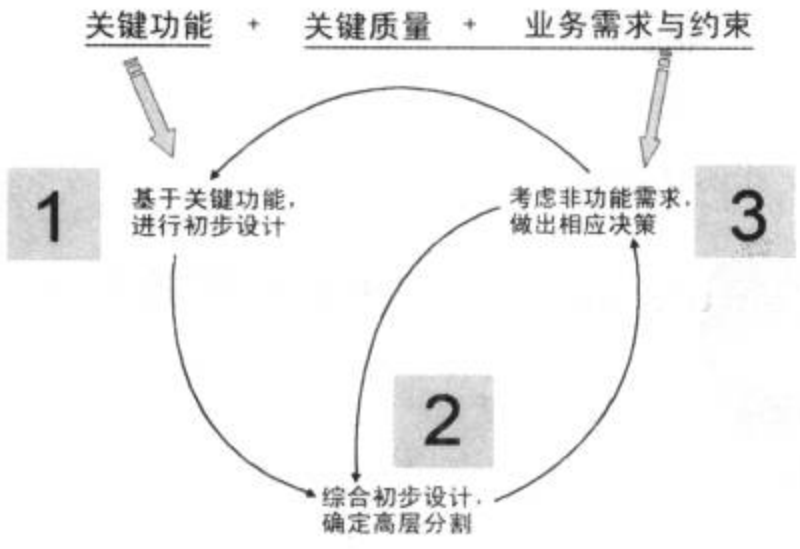


图 1-6 ADMEMS 方法推荐的概念架构设计做法

1.3.3 Refined Architecture 阶段：落地的 5 视图方法

细化架构是相对于概念架构而言的。细化架构阶段的总体方法为 5 视图方法，如图 1-7 所示。

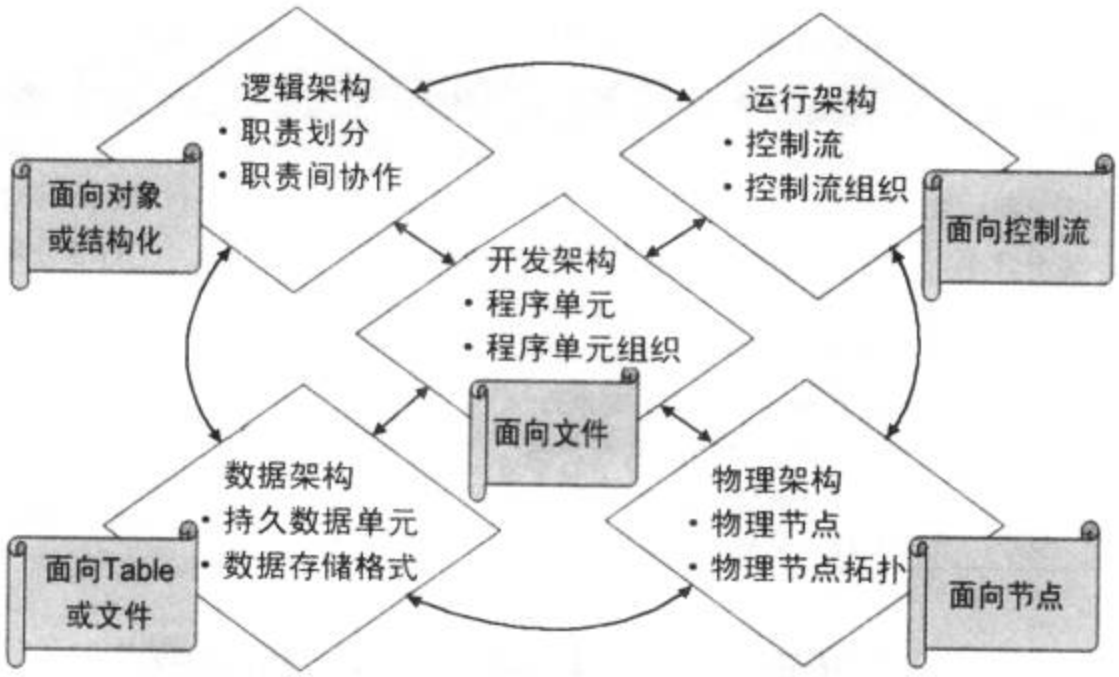


图 1-7　5 视图法：ADMEMS 方法的一部分

许多架构师，言架构则必谈 OO。在他们的思想里，认为 OO 方法已完整涵盖了架构设计的所有方法和技巧。这种看法是相当片面的。

分析图 1-7。若 OO 方法已涵盖架构设计的全部，那么 5 视图方法所涉及的逻辑架构、物理架构、开发架构、运行架构、数据架构，都应全面受到 OO 方法的指导，然而实际上并不是这样。正如图中所标明的，物理架构、开发架构、运行架构和数据架构这 4 个架构视图，分别是面向节点、面向文件、面向控制流和面向 Table（或文件）的——也就是说，一般认为这 4 个架构视图主要的思维并非 OO 思维。另一方面，即使是逻辑架构的设计，也未必都是以 OO 方法为指导的。例如，大量嵌入式软件和系统软件还是以 C 语言为主要开发语言，其逻辑架构设计会以结构化方法为指导。如此看来，倒是将逻辑架构设计总结为“面向职责”更贴近本质。

1.3.4　持续关注非功能需求：“目标-场景-决策”表方法

非功能需求不可能是“速决战”，连编码都会影响到性能等非功能属性，更何况概念架构设计和细化架构设计。

作为 ADMEMS 方法应对非功能需求的思维工具，目标-场景-决策表将架构师的思维可视化了。例如，如表 1-2 所示的目标-场景-决策表，揭示了大型网站高性能设计策略背后的理性思维。

表 1-2 目标-场景-决策表：揭示大型网站高性能设计策略背后的理性思维

目标	场 景	决 策
性能	• 客户端，重复请求页面，Web 服务器请求数多负载压力大	代理服务器
	• 客户端，重复请求页面，页面生成逻辑重复执行	Html 静态化
	• 客户请求，来自不同 ISP，页面跨网络传递慢	内容分发网络
	• 客户端，大量请求图片资源，Web 服务器压力大 • 客户端，大量请求图片资源，Web 服务器无法专门优化	图片服务器
	• 程序，大量申请数据，硬盘 IO 压力大 • 程序，申请不同数据，DBMS 缓存低效	数据库拆分
	• （环境：部署多个 DBMS 实例） 程序，更新数据，数据复制开销大	数据库读写分离

1.4 如何运用本书解决“6 大困惑”

至此，我们已走马观花地了解了本书要讲的 ADMEMS 方法的特点。那么，如何运用本书解决章首提到的“6 大困惑”呢？

如图 1-8 所示，针对 6 个困惑分别给出了阅读路线图。

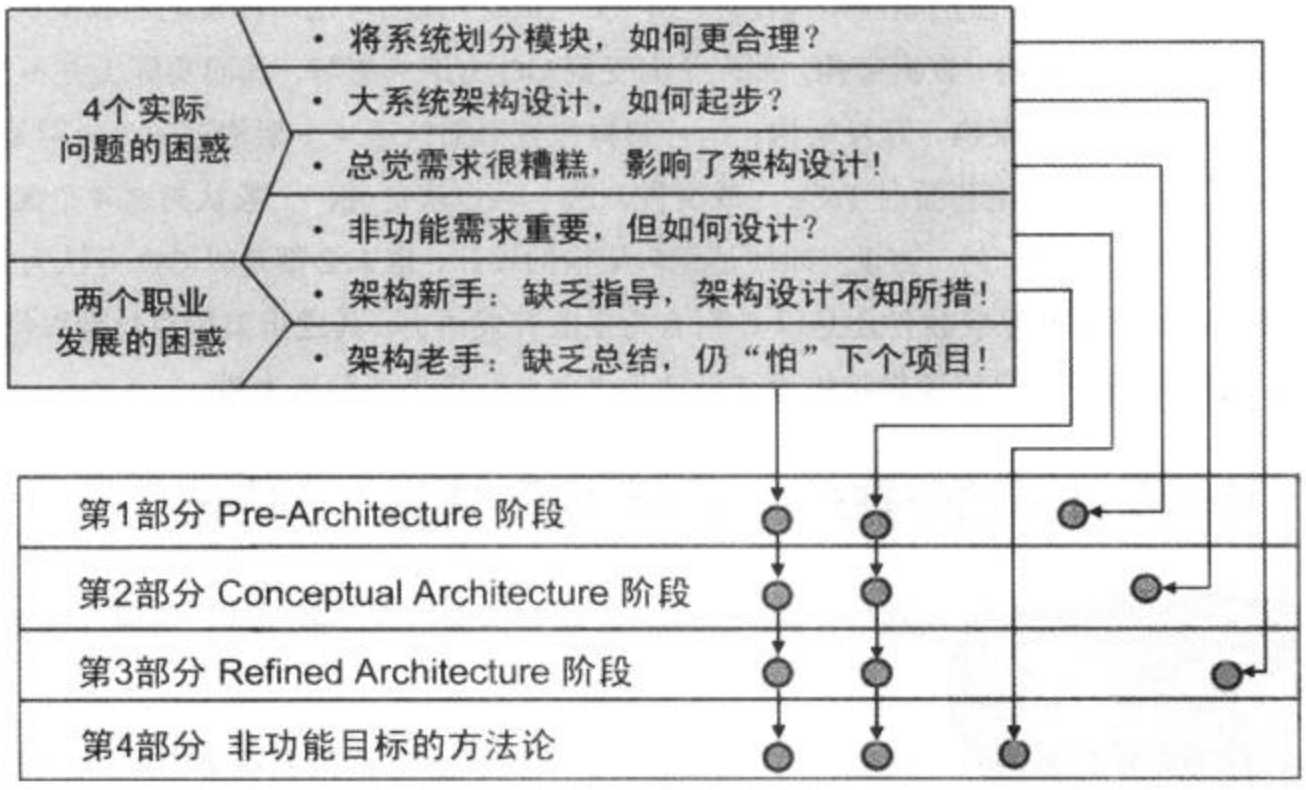


图 1-8 针对 6 个困惑，不同的阅读路线图

例如，如果你是一个已有一定实践经验的架构师，希望更加合理地对系统进行模块切分，请关注“第 3 部分 Refined Architecture 阶段”。你将了解到，划分子系统的 4 大原则（如图 1-9 所示）：

- 职责分离原则。
- 通用专用分离原则。

- 技能分离原则。
- 工作量均衡原则。

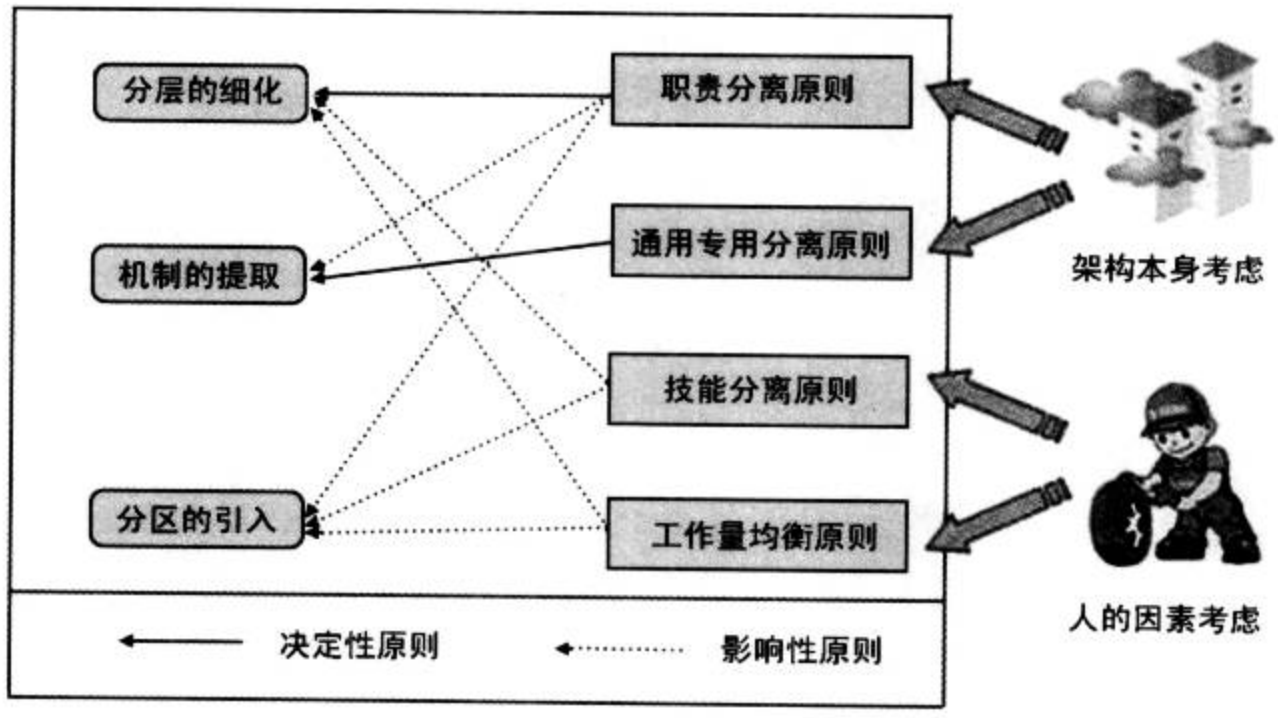
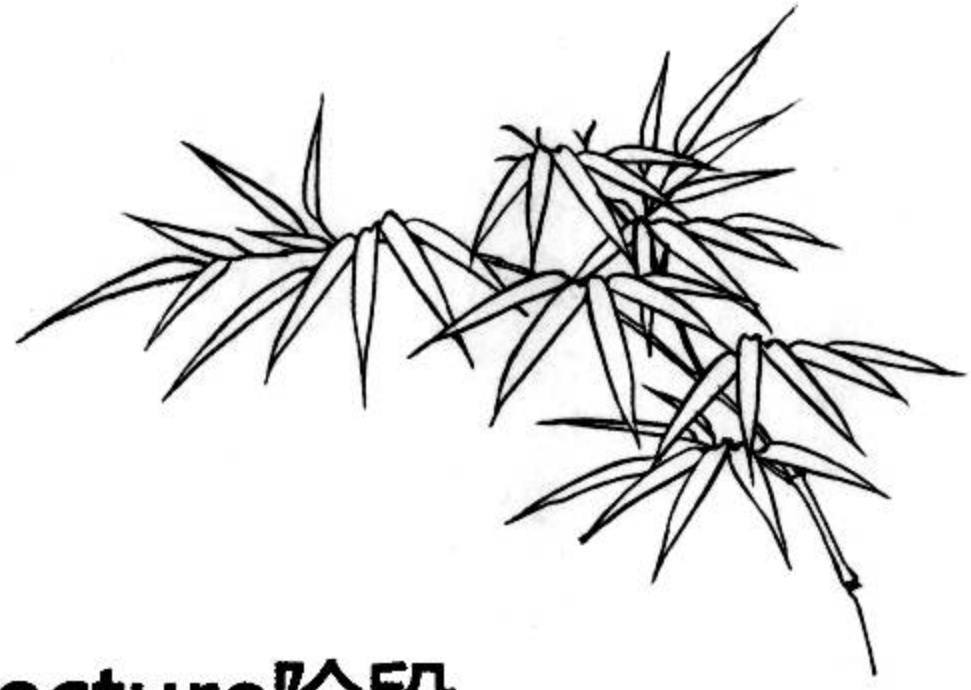


图 1-9 第 3 部分 Refined Architecture 阶段：划分子系统的 4 大原则

其他问题的解决思路在此不再展开叙述，请大家参照“路线图”阅读相应章节。



Pre-Architecture阶段

第2章
Pre-architecture 的故事

所谓的“开始就是结局”，要达到什么样的结局取决于什么样的开始。结局就是开始的地方。（What we call the beginning is often the end. And to make an end is to make a beginning. The end is where we start from.）

——T. S. 艾略特，《四个四重奏》

需求验证的目标是尽可能暴露问题，而不是证明无错。

——徐锋，《软件需求最佳实践》

没有风险的软件早就被开发完了。

作为架构师，首先要面对的风险就是需求。既要关注功能需求，又要平衡相互矛盾的质量属性需求，还不能遗漏各方面的约束性需求……这，已成为合格架构师必需的基本功。

下面的3个真实故事都说明了这一点。

2.1 “不就是个MIS吗”

2.1.1 故事：外籍人员管理系统

公司接单了，一个市级的外籍人员管理系统。

小周被任命为这个项目的架构师。需求分析的过程，小周也参与了。几天之后，小周就开始“轻敌”了，他在一次项目会上说了这么一句话：“这个项目不就是个MIS吗！”

接下来的工作比较顺利，项目组也算情绪高昂……

项目组的情绪急转直下，出现在项目接近尾声的一天。客户方的小崔，看着漂亮的“外籍人

员信息录入”界面，弱弱地说了一句：“哦，外籍人员的信息，大部分都不是我们录入的，而是来自省局。”

这下问题大了。最棘手的问题是，项目定义的数据库 Schema 和省级系统的数据库 Schema 不一致：

- 若保持不一致状态，就人为造成了数据格式的不相容，这是典型的烟囱式应用的做法，为未来可能出现的更多整合要求埋下了障碍；
- 若参考省级系统的数据模型重新定义本系统的数据库 Schema，大量代码就必须重写，项目工期必然拖延。

拼命加班……

2.1.2 探究：哪些因素构成了架构设计的约束性需求

有人说：“错”的一半是“金”，“败”的一半是“贝”。

故事中暴露的问题看似简单：太大意了，遗漏了重要的约束性需求。但试问：下次如何避免？……只有我们这样问自己，才算得上“败”中求“贝”。

请读者也想一想这个问题，笔者推荐的反思结果见于第 4 章“ADMEMS 方法的‘约束分类理论’”一节。

2.2 “必须把虚存管理剪裁掉”

2.2.1 故事：嵌入式 OS 的剪裁

系统软件研究室的新任务启动了：对 VxWorks 操作系统进行剪裁，并开发专用硬件的驱动程序。

团队成员都挺提劲儿……

这天，总工程师要听听进展情况，亲临研究室。当小吴开始汇报对 OS 虚存管理的“深入理解”时，总工程师的表情有些不自然。不久，他打断了小吴，说了这么一句话：“整个系统才有多大内存可用？我们的 OS 占的内存越多，应用软件可用的内存就越少。所以，必须把 OS 的虚存管理剪裁掉，直接访问物理内存。”

举“组”震惊，却又深表折服。

2.2.2 探究：又是约束

架构设计不仅要考虑支持功能、满足质量要求，还要重视各种约束性需求。本例的“内存有

限”，就是嵌入式系统设计中常见的约束。

关注约束，要趁早。

2.3 “都是C++的错，换C重写”

2.3.1 故事：放弃C++，用C重写计费系统

老郑曾经很开心。

老郑在某电信软件企业，负责计费系统的架构。最初，他非常重视系统的性能问题，因为他认为：电信领域用户群广，数据量大，所以性能的压力必然会很大。后来，他们用C++开发的计费系统上线了，用户反映不错，性能挺高的。

但现在，老郑很懊恼。

原因何在呢？原来，计费系统一直面临着功能不断改进的压力，整个团队不断致力于提高系统的可扩展性——以便于增加和修改功能。但始料未及的是：进行了一番“改进”之后，可扩展性上去了，性能下来了！

看着程序里到处都是接口和无处不在的间接，老郑产生了一个危险的念头：都是C++的错，应该用C重写计费系统！

2.3.2 探究：相互矛盾的质量属性

思考如图2-1所示的题目。

互动问答

问题：某公司拟以C语言代替C++，重写其他电信计费系统，因为开发人员引入了太多抽象，使得“可扩展性上去了，性能下来了”。此法可行否？

A.能解决问题

B.于事无补，因为用C语言也会过度设计

C.问题根源出在Pre-architecture

D.架构设计中必须分析质量间相互影响，制定权衡取舍策略

图2-1 思考题

正确答案是：B、C、D。

高性能和灵活扩展这两个质量属性之间存在矛盾关系，这就是要害。表2-1揭示了更多质量属性之间的“促进”或“矛盾”关系，我们吃惊地看到：性能和安全性，与其他许多质量属性都

是矛盾的。

表2-1 质量属性关系矩阵

	持续可用性	性能	可扩展性	安全性	可互操作性	可维护性	可移植性	可靠性	可重用性	鲁棒性	可测试性	易用性
持续可用性								+		+		
性能			−		−	−	−	−		−	−	−
可扩展性		−		−		+	+	+			+	
安全性		−			−				−		−	−
可互操作性		−	+	−			+					
可维护性	+	−	+					+			+	
可移植性		−	+		+	−			+		+	−
可靠性	+	−	+			+				+	+	+
可重用性		−	+	−	+	+	+	−			+	
鲁棒性	+	−						+				+
可测试性	+	−	+			+		+				+
易用性		−								+	−	

（“+”表示行促进列，“−”表示行影响列，“ ”表示行列两种质量属性之间影响不明显）

本书“Pre-architecture阶段篇”后续内容提供的正确做法是：

1. 在架构设计之初，就全盘考虑架构设计要重点支持的关键质量目标——以老郑为例，性能和可扩展（当然还会有其他质量属性）都重要。
2. 在第一时间就判断这些“关键质量”之间有没有冲突关系，并制定权衡取舍策略——仍以老郑为例，性能和可扩展性矛盾，性能的优先级更高，谨慎评审提高可扩展性的设计对性能的影响后决定是否采用。

2.4 展望“Pre-architecture阶段篇”

失败的故事，何止3个？透过这冰山一角，我们看到的是一线架构师“把握需求技能的缺失”。

软件架构师不必是需求捕获专家，也不必是编写《软件需求规格说明书》的专家；但他一定应在需求分类、需求折衷和需求变更的研究方面是专家，否则他和优秀软件架构师相比就输在了“起跑线”上。

本书后续几章是Pre-architecture阶段的内容展开部分，将讲解如何建立需求理解的大局观、如何把握需求特点、确定架构设计驱动力。

第3章
Pre-architecture 总论

凡事预则立，不预则废。

——孔子，《礼记·中庸》

业内对架构的讨论仍沿用了传统思想：如果知道了系统需求，就可以为此系统构建架构。这种观点是缺乏远见的……

——Len Bass，《软件构架实践（第2版）》

架构设计对系统成败非常关键，那么，什么对架构设计的成败非常关键呢？

功能需求、质量属性及约束共同决定了架构，对这3类需求的把握是否到位、设计决策是否对路，是架构设计成败的关键所在！

然而，业界的现状却是：

- “架构师就是技术高手”的声音充耳不绝，我们常错误地认为“架构师不必懂需求”。例如，许多架构师不知业务级需求、用户级需求、开发级需求包含的具体内容，更不懂功能、质量、约束对架构设计的影响如何大相径庭。
- 只有少数经验深厚的架构师在“拿到”需求之后，会基于业务背景、系统规模、技术趋势、开发团队现状等现实情况，对需求进行理性的、有针对性的权衡、取舍、补充。而在方法一级，Pre-architecture 阶段基本是空白。

对此，本书希望在方法一级为“Pre-architecture 阶段”提供较明确的指导——以 ADMEMS 矩阵为核心的“四步法”：

1. 需求结构化
2. 分析约束影响
3. 确定关键质量
4. 确定关键功能

3.1 什么是 Pre-architecture

本书认为，完整覆盖“需求进，架构出”的架构设计方法才是符合一线实践需要的。

Pre-architecture 就是架构设计的最前期阶段，其工作目标包括：理解需求、建立需求大局观、确定架构设计方向等。

“磨刀不误砍柴工”，这是近乎常识的古训。整个 ADMEMS 方法包含 Pre-architecture、Conceptual Architecture、Refined Architecture 3 个阶段。如果说，Conceptual Architecture 和 Refined Architecture 阶段是“砍柴”（这两个阶段都对系统进行了某种程度的切分，系统已经不再是“黑盒子”了），那么最初的 Pre-architecture 阶段就是“磨刀”（此阶段未对系统进行切分，系统还是“黑盒子”）。

3.2 实际意义

Pre-architecture 阶段虽然是铺垫性质的阶段，但对架构实践而言意义重大。

3.2.1 需求理解的大局观

架构师常常面对相互矛盾的需求目标。如果对需求的理解缺乏大局观，那将如何进行需求的权衡取舍？

重大需求塑造概念架构。如果对需求的理解缺乏大局观，那将如何识别重大需求、特色需求、高风险需求？

架构师必须在相对短的时间内设计架构。如果对需求的理解缺乏大局观，那将如何进一步运用“关键需求决定架构，其余需求验证架构”的策略？

Pre-architecture 阶段能帮助架构师建立需求理解的大局观。任何需求都可定位于业务级需求、用户级需求、开发级需求这 3 个需求层次的某一层，同时也必属于功能、质量、约束这 3 类需求的某一类。如此一来，就便于梳脉理络、把握关系。

3.2.2 降低架构失败风险

对需求理解不透、遗漏需求往往是架构设计失败的重要原因（如图 3-1 所示）。在你的身边，一定有许多类似上一章 3 个真实故事那样的案例。

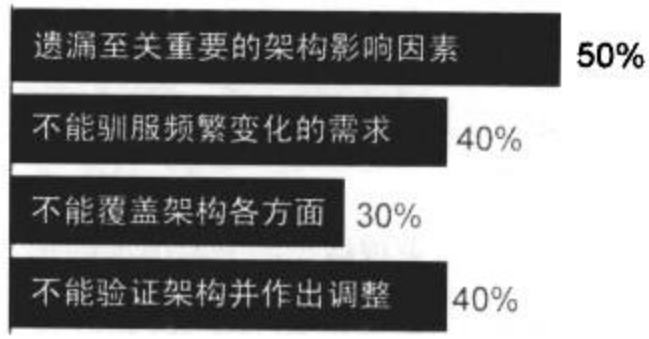


图 3-1 架构设计失败的常见原因

有一幅软件行业自嘲的漫画（如图 3-2 所示），讲的是猴子希望得到一串香蕉，收到的却是骨头——这个礼物并不能满足它的真实需求。相信许多人看到这幅漫画会苦笑不已。用户经常得不到真正满足他们需求的系统，这已成为整个软件业界一个严峻的问题。

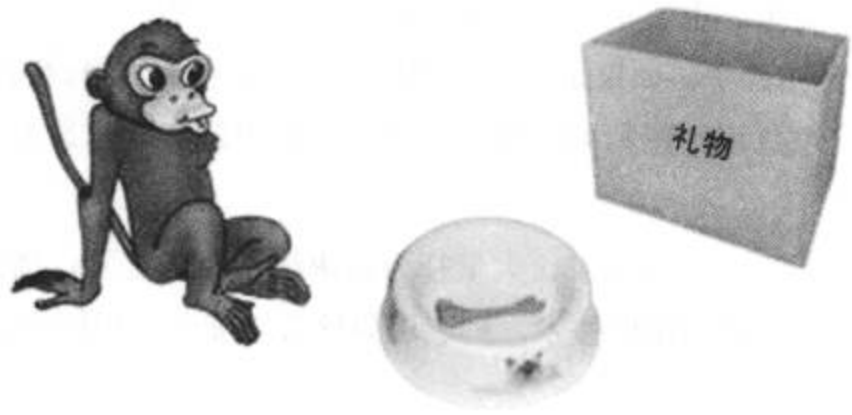


图 3-2 漫画：用户得不到真正满足他们需求的系统

我们无法回避一个事实：架构师在需求的理解、权衡、取舍和补充这几方面能力严重不足，是造成猴子得到骨头的重要原因之一。如果你对此有异议（认为那完全是需求分析师的责任），请看《软件工程的事实与谬误》中“事实 26”：

从需求转入设计时，因为制定方案过程的复杂性，会激增出大量的衍生需求（针对一种特定设计方案的需求）。设计需求是原始需求的 50 倍之多。

ADMEMS 方法的 Pre-architecture 阶段将告诉一线架构师，如何告别拙劣的“需求列表”方式（它难道不是遗漏架构影响因素的罪魁祸首吗？），取而代之以 ADMEMS 矩阵方法。的确，需求是有结构的！由于 ADMEMS 的“二维需求观”体现了更复杂的、更本质的需求结构，所以可以帮助架构师更全面地看待需求、避免遗漏重要的非功能需求，大大降低架构失败的风险。

3.2.3 尽早开始架构设计

如何尽早开始软件架构设计？这是很多软件企业非常关心的一个问题，因为大家都深感工期的巨大压力。

灵活运用 Pre-architecture 阶段的方法，有一个额外的好处：能够在需求没有“全面完成”的情况下就开始架构设计。具体而言，为了尽早开始架构设计，软件企业必须做好以下两点：

- 让架构师参与需求分析工作。

实际上，是让架构师相对自由地“全程”参与需求分析工作——甚至可以不为任何具体的需求捕获、需求分析、需求文档工作负责。

- 不能被动地等待完善的《软件需求规格说明书》出现的那一刻。

建议架构师在参与需求分析工作时，不断运用 ADMEMS 矩阵等工具对需求进行大局的梳理，只要满足下列 3 个条件（如图 3-3 所示）就可以尽早开始架构设计工作：

1. 有了明确的业务需求。必须保证甲、乙双方就建设系统的目标（可能不止一项）达成了共识，《愿景文档》经过了正式评审，并且明确了投资、工期标准、整合等约束条件。试想，业务需求含糊不清，架构设计方向如何确定呢？
2. 了解全面的用户需求。也就是说，系统能帮助用户干什么、不能干什么，这个“需求的 Scope”已经非常明确了。如果采用了用例技术，则表现为“用例图”是比较完善的，没有明显的遗漏（注意用例图和用例规约在需求分析中的实践意义不同，可参考《软件架构设计》一书）。
3. 有了典型的行为需求。这意味着，大量行为需求还未明确定义，离提交《软件需求规格说明书》还早。如果采用用例技术，则表现为核心功能的《用例规约》已定义。

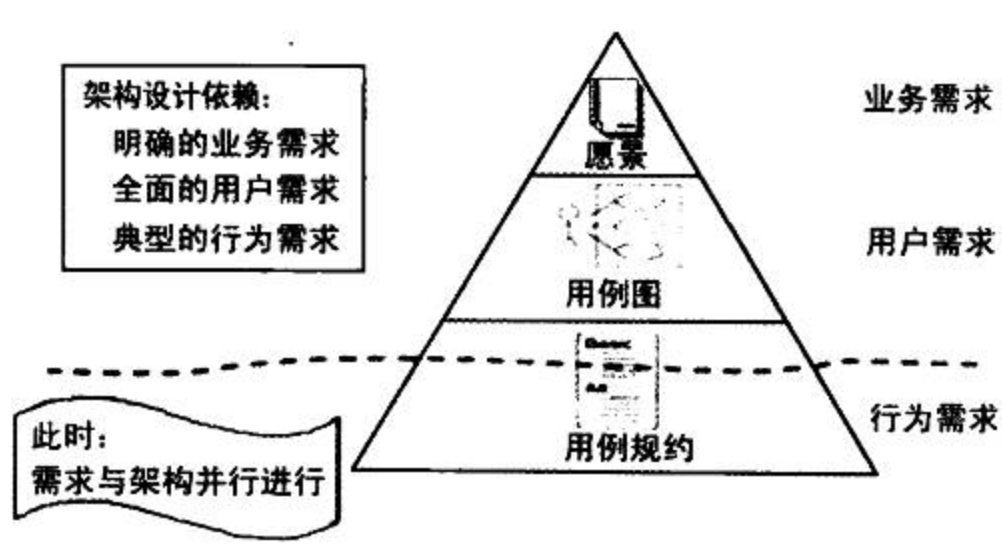


图 3-3 尽早开始架构设计的 3 个条件

相反，连上述 3 个“最基本条件”都不具备就开始架构设计是盲目的。

3.2.4 明确架构设计的“驱动力”

架构设计的“驱动力”不就是《软件需求规格说明书》吗？这种观点，只对了一半。

问题 1：试想，《软件需求规格说明书》中几乎没有定义非功能需求（客户方也签字了），架构师就可以不考虑非功能需求了吗？

问题 2：试想，需求变更难以避免，如果以所有需求作为架构设计的“驱动力”会是什么结果？

问题 3：试想，《软件需求规格说明书》中照抄了《ISO 9126》中的所有质量属性要求，架构师应该不计成本、不分重点地全部支持吗？

上述 3 个一线架构设计中常见的问题，都说明架构师除了需要关注《软件需求规格说明书》之外，还必须关注其他很多因素，最终或添、或减、或折衷，理性地确定真正的架构设计“驱动力”。

具体而言，Pre-architecture 阶段将告诉我们不辱使命的方法：

- 分析业务需求和约束背后的衍生需求——针对问题 1。
- 发现遗漏需求——针对问题 1。
- 确定关键功能——针对问题 2。
- 确定关键质量——针对问题 2。
- 权衡质量属性之间的矛盾关系——针对问题 3。

3.3 业界现状

既然 Pre-architecture 阶段如此重要，业界现状如何呢？很遗憾，业界对 Pre-architecture 阶段普遍不够重视。相反，“架构设计唯靠经验”、“架构设计目标不变”等错误观点比较常见。

我们头脑中的“位置”是有限的，如果错误的认识占据了主导位置就会导致实践偏差（有兴趣的朋友可以读一读关于“心智模式”的书），因此必须防止。

3.3.1 “唯经验论”

和“架构师不必懂需求”的误解相比，“唯经验论”已经有所进步了。这种观点认为，架构师纯粹凭借经验，发现需求的遗漏、权衡需求之间的矛盾、确定架构设计的重点目标。

必须承认，经验对架构设计很重要，但“唯经验论”依然是错误的。

因为，世界上并不存在两个完全相同的项目，不同项目在功能需求、质量属性，以及约束这 3 方面必然存在差异。于是，架构师不仅应具有一定经验，还必须掌握超越具体项目的、更具普遍意义的方法与技能。

3.3.2 “目标不变论”

架构设计的目标不是一成不变的，基于此认识，ADMEMS 方法在 Pre-architecture 阶段的“确定关键质量”环节提供了专门的指导。

“架构设计目标不变论”是错误的。例如，有网友认为：

我们必须牢记架构设计的总体目标，可以概括为下面几点。

1. 最大化地重用……
2. 尽可能地简单明了……
3. 最灵活的拓展性……

首先，若架构设计的目标真能概括为不变的“几点”，那可算是架构师的福音了，但实际上，架构设计的目标必然随领域不同（如航空航天、电信、电子政务）、规模不同（如项目、产品、平台）、条件不同（如工期、预算、标准）而变化。

其次，为重用、简单、可扩展都加了“最”（而不是权衡折衷），不符合架构设计的现实，更何况“灵活”和“简单”之间常常存在矛盾。

3.3.3 需求分类法的现状

软件行业处在不断的发展变化之中，软件需求分类法就是一例。当前，业界影响最为广泛的需求分类法将需求分为 3 个层次，如图 3-4 所示。

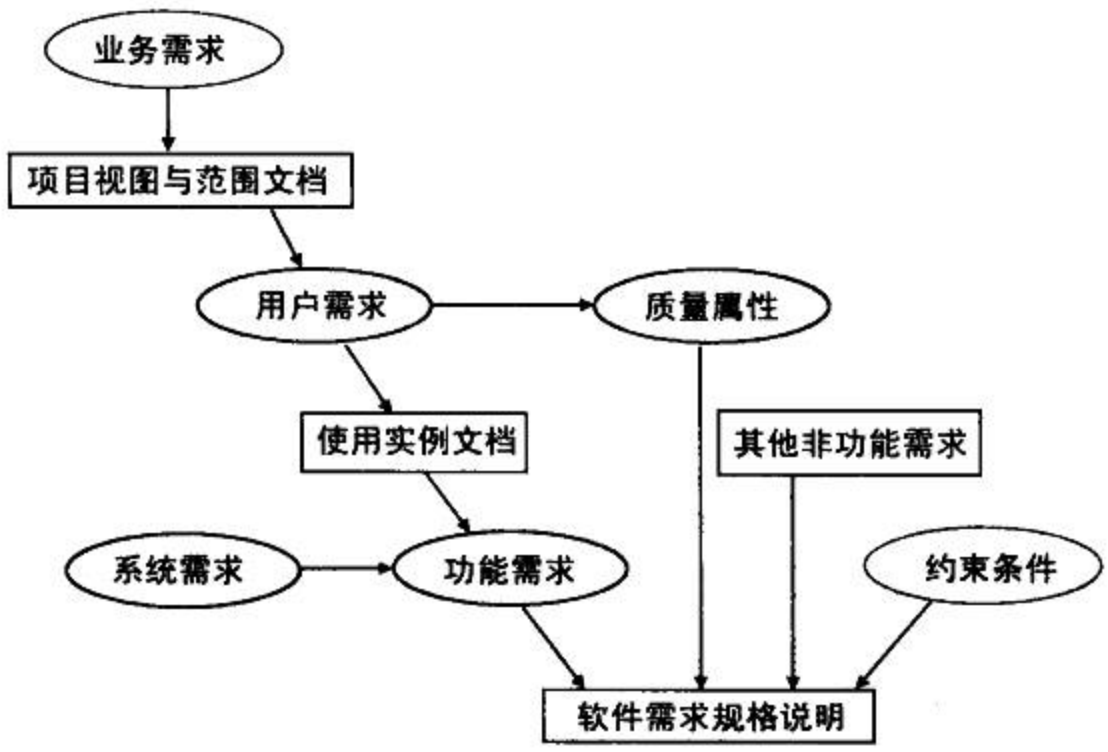


图 3-4 需求的 3 个层次（图片来源：《软件需求》）

简评几句。这种需求分类法最大的好处是明确了不同层次需求之间的跟踪关系——业务需求—>用户需求—>行为需求（功能需求另一个更好的名字）——从而建立了需求分析的主要脉络，非常有意义（事实也证明了这一点）。但是，对架构师而言，这种需求分类法中的“约束条件”太过狭隘了，没有反映“架构设计必须面对来自业务环境、使用环境、构建环境、技术环境的 4 大类约束”这一现实情况。

再例如，RUP 提倡的需求分类法也包含 3 个层次：需要（Need）、特性（Feature）、软件需求。如图 3-5 所示。

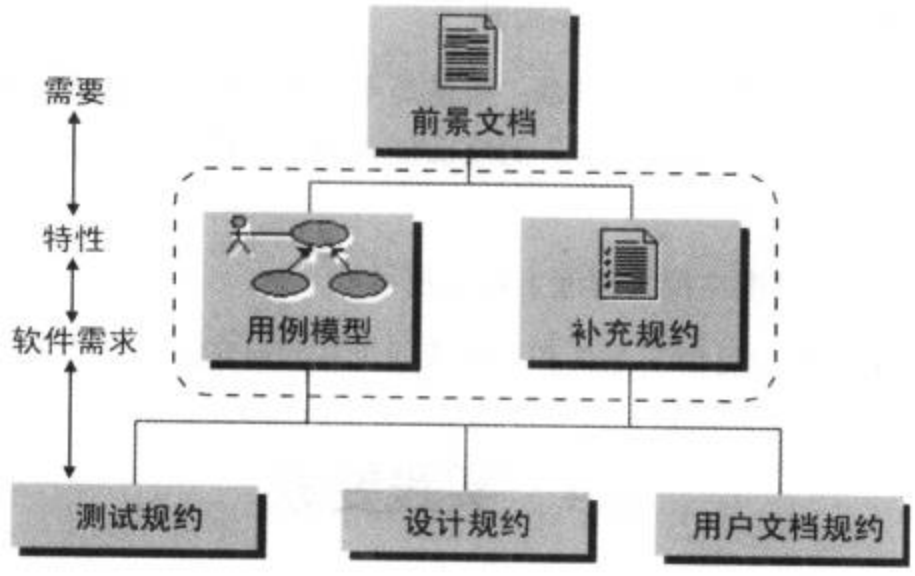


图 3-5 RUP 提倡的需求结构（图片来源：RUP）

简评几句。这种分类法也是主要为需求分析工作服务的，它除了非常倚重用例技术外，还有一个明显特点——Feature。以特性（Feature）技术作为从需要（Need）向软件需求过渡的跳板，是解决需求分析中“从需要向软件需求跨度过大”问题不错的选择，这一点已受到许多实践者的认同。缺点嘛，用例的地位过分突出了，又由于“用例涉及但不涵盖非功能需求”的性质，不少一线实践者遗漏非功能需求的常见问题也就不难解释了。

最后，顺便指明上述两种“需求层次论”的对应关系，此问题令颇多实践者困惑。如图 3-6 所示。

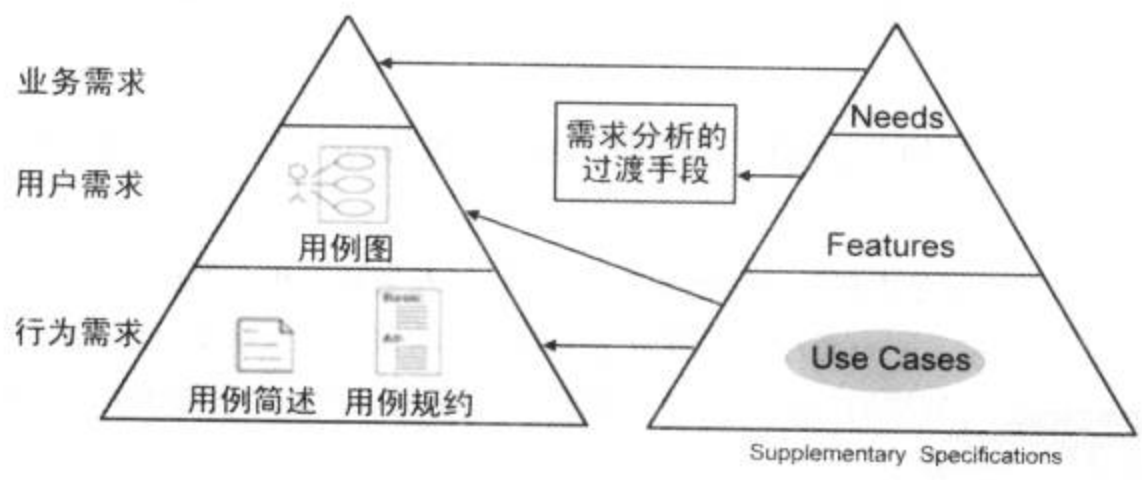


图 3-6 两种“需求层次论”的对应关系

3.3.4 需求决定架构的原理亟待归纳

不懂不同需求分别如何影响架构，就难以进行理性的架构设计，难免“拍脑袋”决策。

关于需求决定架构原理，业界当前的认识状况如下：

- 功能影响架构的原理，研究得最透彻。

我们可明确将之归纳为：功能得以完成是依赖“职责协作链”，反过来针对功能规划“职责协作链”有助于推动架构设计。

- 质量影响架构的原理，也有基本共识。

但是，还缺乏动态考虑。本书的归纳是：质量是不可能“抛开”功能单独决定架构的，因为质量的考虑是通过对“架构设计中间成果”的质疑展开的，以推动对“架构设计中间成果”的细化、调整，甚至推倒。

- 约束影响架构的原理在很大程度上被忽视了。

本书将提出“约束的4大类型”理论和“约束是架构设计的上下文”观点。

3.4 实践要领

问题是方法之父。不怕有问题，就怕发现不了问题。

对于在“业界现状”一节所列的 Pre-architecture 阶段存在的“问题”，当如何解决呢？下面简明扼要地予以说明。

3.4.1 不同需求影响架构的不同原理，才是架构设计思维的基础

当前业界，大多数架构师都认同“需求决定架构”，但对需求“如何决定”架构还知之不深。

请每个架构师问自己这样一个问题：需求决定架构，真的这么简单吗？

倘若真这么简单，为何“我”常常对需求已“心知肚明”，却依然对架构设计“一筹莫展”呢？

……

答案是：“需求决定架构”是对的，但不同需求影响架构的不同原理，才是架构设计思维的基础。

哲学家培根（“知识就是力量”就是他说的）指出：“人类知识和人类权力归于一；因为凡不知原因时即不能产生结果。……而凡在思辨中为原因者在动作中则为法则。”这段话意思很明白：架构师不懂得“需求决定架构的原理”这个“原因”，在架构设计这个（思维）“动作”中就没有“法则”呀！

表 3-1 归纳了不同需求是如何以不同原理影响架构设计的。

表 3-1 不同需求，影响架构的原理不同

需求	基本原理	对架构设计的影响
功能	功能是发现职责的依据	• 每个功能都是由一条“职责协作链”完成的，架构师通过为功能规划职责协作链、将职责分配到子系统、为子系统界定接口、确定基于接口的交互机制，来推动架构设计的进行
质量	质量是完善架构设计的动力	• （必须）基于当前的架构设计中间成果，进一步考虑具体质量要求，对架构设计中间成果进行细化、调整、甚至推倒重来，一步步地使架构设计完善起来 • 质量和功能共同影响着架构设计，抛开功能、单依据质量要求设计架构是不可能的

（续表）

需求	基本原理	对架构设计的影响
约束	约束对架构设计的影响分为几类	• 直接制约设计决策的约束（例如“系统运行于 Unix 平台之上”） • 转化为功能需求的约束（例如“本银行系统执行现行利率”引出“利率调整”功能） • 转化为质量属性需求的约束（例如“柜员计算机平均水平不高”引出易用性需求）

任何一项功能都是由一条特定的“职责协作链”完成的，如图 3-7 所示。作为完整的软件系统，它在支持每一个具体功能时，都必然涉及软件不同“部分”之间的相互配合。系统的控制权在这些不同的“部分”之间来回传递，形成一条“职责协作链”，可以完成非常复杂的功能。

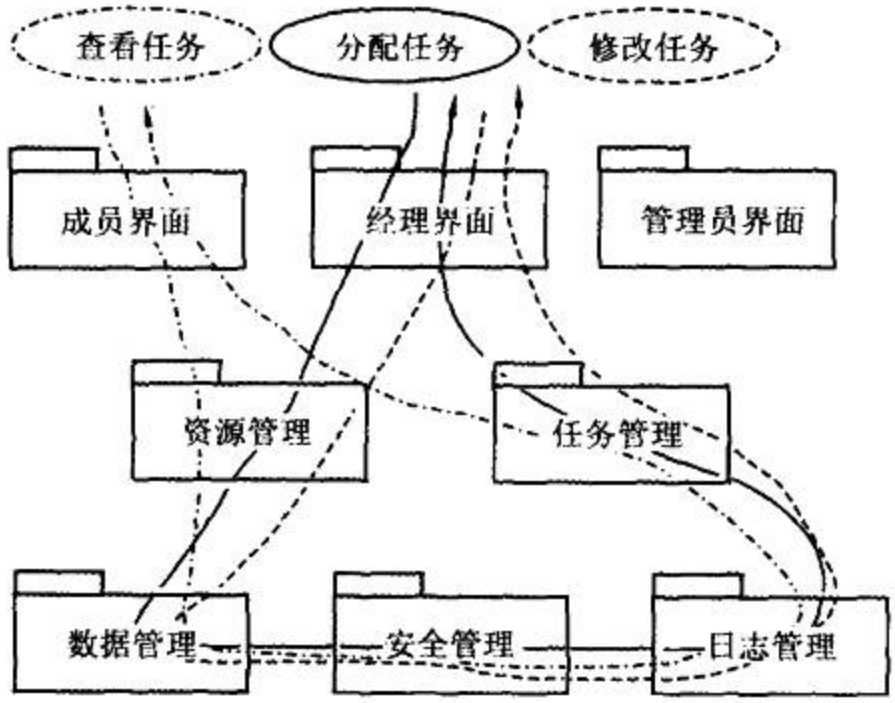


图 3-7 功能影响架构的基本原理：职责协作链

而质量，是完善架构设计的驱动力，不考虑质量的系统是无法走出实验室的。如图 3-8 所示，基于中间设计成果进一步质疑是其中基本的“思维方式”。例如，如果只考虑功能，“页面缓存”的设计就永远不会被引入，它是质疑性能、调整设计的结果。

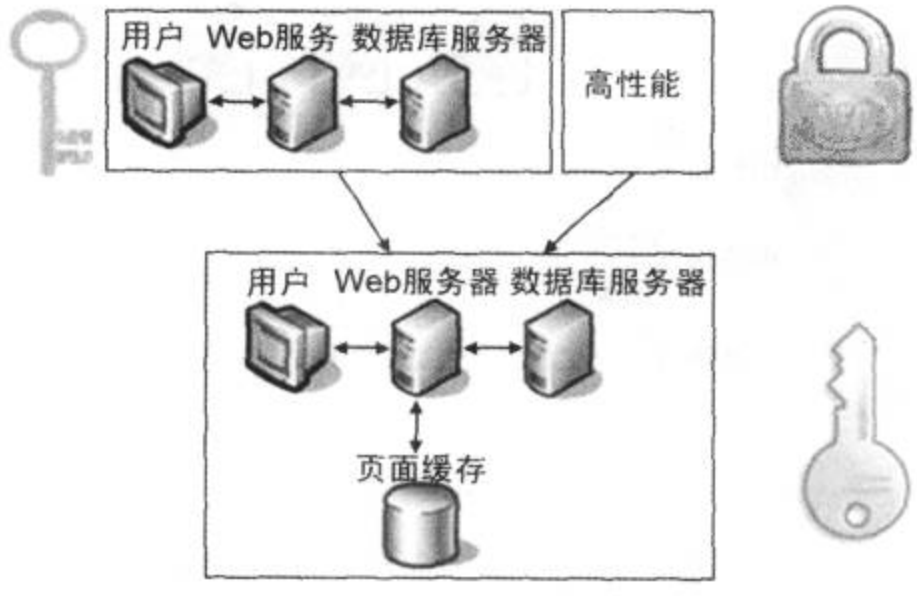


图 3-8 质量影响架构的基本原理：进一步质疑

至于约束，则有不同的具体方式影响着架构设计（结合表 3-2 的示例进行说明）：

- 直接制约设计决策的约束。例如，“系统运行于 Unix 平台之上”作为一条约束，架构师直接遵守即可。
- 转化为功能需求的约束。例如，“本银行系统必须严格执行人民银行统一规定的利率”是一条约束，但分析后发现，正是它引出了一条功能需求：即必须提供调整利率设置的实用功能。
- 转化为质量属性需求的约束。例如，有经验的系统分析员发现了一条重要约束：“任职于各储蓄所和分理处的柜员，计算机水平普遍不高”。由此，未来的软件系统必须具有很高的易用性（否则不会用）和鲁棒性（否则可能把系统搞瘫痪了）就是非常必要的。

表 3-2 银行系统：分析约束影响示例

	功能	质量	约束
业务级需求	• 适应业务变化	• 4 个月就交付上线	• 严格执行人行统一规定的利率 • 系统运行于 Unix 平台之上 • 与原有行长办公系统集成
用户级需求	• 利率调整功能	• 易用性 • 鲁棒性	• 任职于各储蓄所和分理处的柜员计算机水平普遍不高
开发级需求		• 可修改性 • 可扩展性 • 互操作性	

3.4.2 二维需求观与 ADMEMS 矩阵方法

ADMEMS 方法提倡“二维需求观”，如图 3-9 所示。

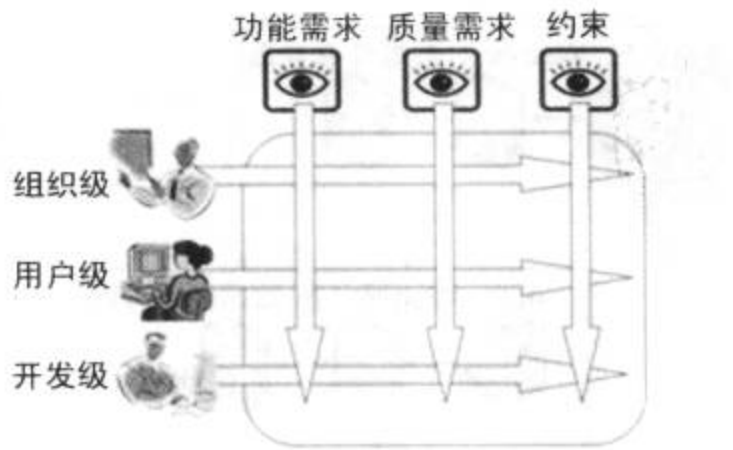


图 3-9 二维需求观

观念是行为的向导，有怎样的观念存在，就有怎样的行为方式产生——突破拙劣观念的意义就在于此。作为架构师，如果在你的观念中，需求是一个散乱无序的“列表”，面对复杂系统时

就会非常被动。需求列表这种贴着“简单”这个“招人待见”标签的方法已经影响、并正在继续影响着许多一线架构师。ADMEMS 矩阵方法的本质，是要突破“需求列表”这种“需求观”对架构设计的束缚。

更多介绍请参考下一章“工具：ADMEMS 矩阵”一节的内容。

3.4.3 关键需求决定架构，其余需求验证架构

有经验的架构师，懂得在实践中运用“关键需求决定架构”的理念。具体而言，可以用 3 句话概括：

- 功能需求做减法。在所有功能中挑选一个“关键功能子集”，作为“架构设计驱动力”的第 1 部分。
- 质量属性需求做减法。根据系统所在领域的特点及系统规模等因素，确定架构设计重点支持哪些质量属性，作为“架构设计驱动力”的第 2 部分。
- 约束性需求做加法。充分考虑需求方及业务环境因素、用户群及使用环境因素、开发方及构建环境因素、业界当前技术环境因素等“4 类约束”，将之作为“架构设计驱动力”的第 3 部分，并以“做加法”的思维分析约束影响、识别约束背后的衍生需求。

3.4.4 Pre-architecture 阶段的 4 个步骤

Pre-architecture 阶段对整个架构设计工作非常重要，它担负着建立需求大局观、把握需求特点、确定架构设计驱动力的责任。但是，Pre-architecture 阶段的步骤非常清晰，如图 3-10 所示。

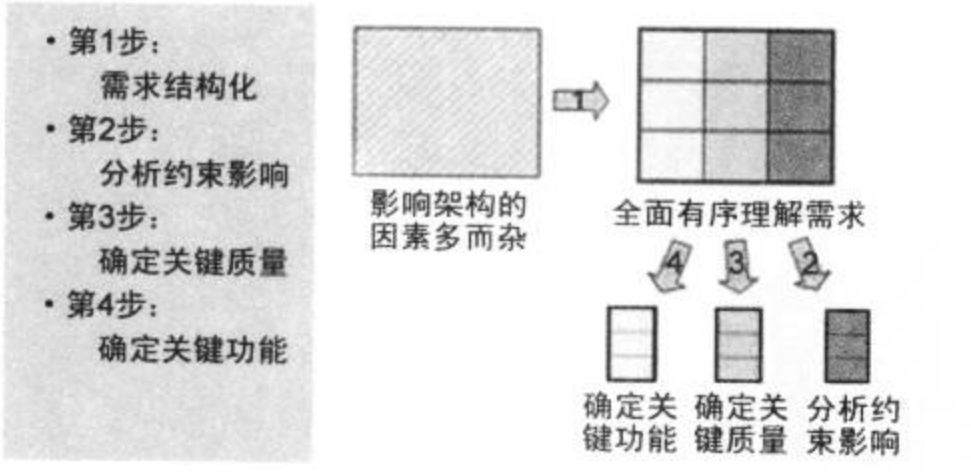


图 3-10 Pre-architecture 阶段的 4 个步骤

首先，借助 ADMEMS 矩阵思维工具，把多而杂的架构影响因素梳脉理络、建立全面有序的理解。

然后，分别针对约束、质量、功能这 3 类需求开展相应工作。分析约束影响，识别隐含需求；确定关键质量，明确关键质量之间的优先级；确定关键功能，便于更有针对性地分配有限的架构设计时间。

第 4 章
需求结构化与分析约束影响

心念不同，判断力自然不同。

——严定暹，《格局决定结局》

全面认识需求，是生产出高质量软件所必须的“第一项修炼”。

——温昱，《软件架构设计》

ADMEMS 方法的 Pre-architecture 阶段包括 4 个步骤，本章讲解前两步：

- 第 1 步，需求结构化。
- 第 2 步，分析约束影响。

4.1 为什么必须进行需求结构化

需求是有结构的。

许多实践者不懂得这一点，更不知如何“主动运用”这一点。在他们眼中，架构设计要应对的需求往往是又多又乱的，而且遗漏了关键需求也发现不了……

相反，有经验的架构师懂得运用需求的结构。他们能够将复杂的需求集合梳理得井井有条，为进一步分析不同需求之间的联系（作为权衡折衷的依据）、识别遗漏的重要需求打下坚实基础。

Pre-architecture 阶段要求进行需求结构化，这代表着 ADMEMS 方法更贴近一线架构设计的真实实践。通过 ADMEMS 矩阵这种思维工具，可以全面理解需求的各个层次、各个方面，更为分析需求之间关系、识别遗漏需求、发现延伸需求奠定基础。通过形象的“物体归类”隐喻（如图 4-1 所示）可以加深对需求结构化工作的理解。

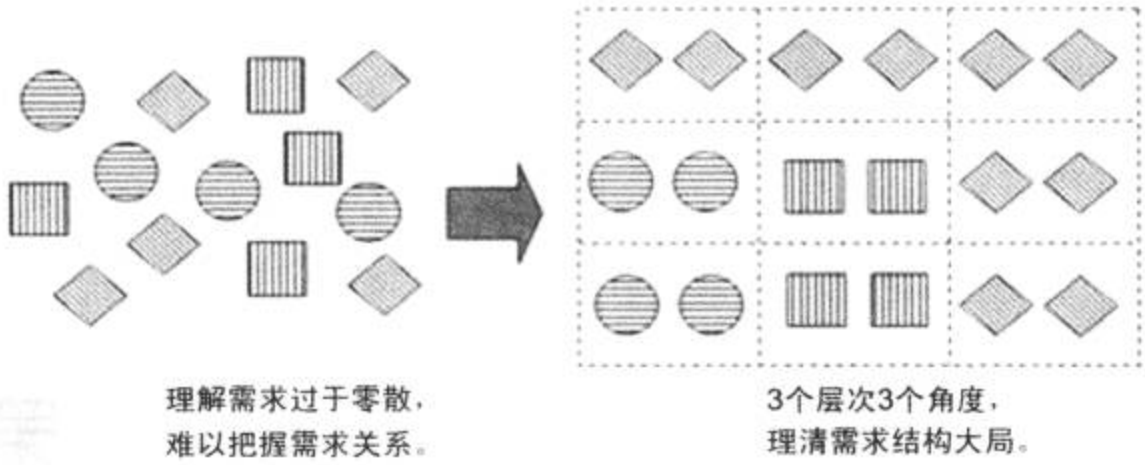


图 4-1 需求结构化的隐喻

4.2 用 ADMEMS 矩阵方法进行需求结构化

那么，需求结构化要怎么做呢？

第一，绝对不能认为《软件需求规格说明书》就是需求的全部。

第二，运用 ADMEMS 矩阵方法。

4.2.1 范围：超越《软件需求规格说明书》

首先，需求文档常常不够全面，所有有经验的架构师都重视需求文档，但不应该“唯需求文档是瞻”。

其次，需求变更经常发生，“依赖且仅依赖需求文档”不够聪明，使架构设计工作非常被动。

既然架构师必须“对需求进行理性的、有针对性地权衡、取舍、补充”，那么“作为架构设计驱动力的需求因素”和“供甲方确认的《软件需求规格说明书》”之间就必然不能“划等号”。

所以，架构师要通过需求结构化真正全面地“鸟瞰”需求大局，就必须超越《软件需求规格说明书》。

还有一个重大意义在于，只有摆脱对《软件需求规格说明书》提交时间、文档质量、内容变更的“呆板依赖”，才有可能尽早开始架构设计（请参考第 3 章中“尽早开始架构设计”一节）。

4.2.2 工具：ADMEMS 矩阵

矩阵，是很多著名方法的核心。例如，制定公司层战略的方法之一是“波士顿矩阵”，“波士顿矩阵”又称“市场增长率—相对市场份额矩阵”。

本书推荐的核心工具之一就是“ADMEMS 矩阵”，它又称为“需求层次—需求方面矩阵”。如表 4-1 所示。

表 4-1 ADMEMS 矩阵（需求层次—需求方面矩阵）

	广义功能	质量	约束
业务级需求	业务目标	快、好、省	技术性约束 法规性约束 技术趋势 竞争因素与竞争对手 遗留系统集成 标准性约束 分批实施
用户级需求	用户需求	运行期质量	用户群特点 用户水平 多国语言
开发级需求	行为需求	开发期质量	开发团队技术水平 开发团队磨合程度 开发团队分布情况 开发团队业务知识 管理：保密要求 管理：产品规划 安装 维护

首先，需求是分层次的。

业务级需求：包含客户或出资者要达到的业务目标、预期投资、工期要求，以及要符合哪些标准、对哪些遗留系统进行整合等约束条件。

用户级需求：用户使用系统来辅助完成哪些工作？对质量有何要求？用户群及所处的使用环境方面有何特殊要求？

开发级需求：开发人员需要实现什么？开发期间、维护期间有何质量考虑？开发团队的哪些情况会反过来影响架构？

可以看出，需求的三个层次，是站在“不同层次的涉众提出需求所站的立场不同”的角度，将需求划分为三种类型。其次，需求还必须从不同方面进行考虑。例如，一个网上书店系统的功能需求可能包括“浏览书目”、“下订单”、“跟踪订单状态”、“为书籍打分”等，质量属性需求包括“互操作性”和“安全性”等，而“必须运行于 Linux 平台之上”属于约束性需求之列。实践一再表明，忽视质量属性和约束性需求，常常导致架构设计最终失败。

于是，从“需求定义了直接目标还是间接限制”的角度，把需求划分为 3 种类型，这就是需求的 3 个方面：

功能需求：更多体现各级直接目标要求。

质量属性：运行期质量 + 开发期质量。

约束需求：业务环境因素 + 使用环境因素 + 构建环境因素 + 技术环境因素。

一句话，需求是有结构的。而且，需求的结构绝对不是“List”，而应该是“二维数组”。

用 ADMEMS 矩阵方法进行需求结构化，非常直观。作为一种思维工具，ADMEMS 矩阵背后的原理是“二维需求观”，这是“需求列表”这种“一维需求观”所不及的。这就好比程序设计选择了不合适的数据结构，那么功能的实现就要多费周折——既然 List、Tree、Graph 等数据结构大家都了解，用此作为类比再合适不过了。

结构化是控制复杂性的好办法。例如一个会计师，如果他的办公桌上凌乱地堆满了曲别针、铅笔、硬币……他会因为“东西多”而怎么也找不到某样东西。相反，将不同物品梳脉理络、分门别类地进行“摆放”，就不会丢东忘西（如图 4-2 所示）。

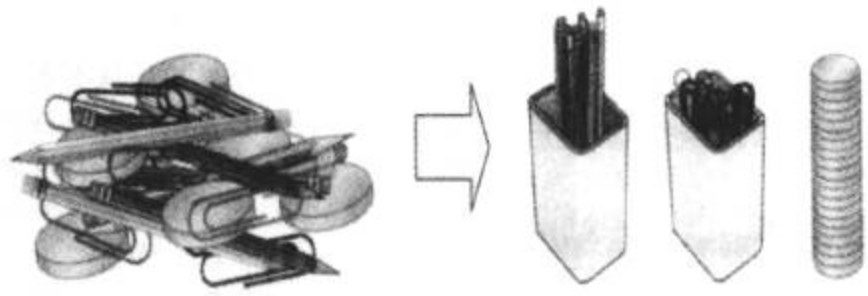


图 4-2 结构化：分门别类地进行“摆放”

很奇妙，进行需求结构化之后，架构师会感觉“需求变少了”——其实，需求没有变少，但复杂性却得到了控制。“需求变少了”的最大好处是，架构师可以比原来轻易得多地发现遗漏需求。

上述方法的运用，请参考本章第 7 节“大型 B2C 网站案例：需求结构化与分析约束影响”。

4.3 为什么必须分析约束影响

风险有个恼人的特点：一旦你忘了它，它就会找上门来制造麻烦。

对于架构设计而言，来自方方面面的约束性需求中潜藏了大量风险因素。所以，有经验的架构师都懂得主动分析约束影响，识别架构影响因素，以便在架构设计中引入相应决策予以应对。

同样，Pre-architecture 阶段明确要求必须分析约束影响。背鳍下面是不是一条鲨鱼？约束性需求中，是不是潜藏了风险因素？如图 4-3 所示的隐喻，点明了分析约束影响的要义：尽早识别风险。

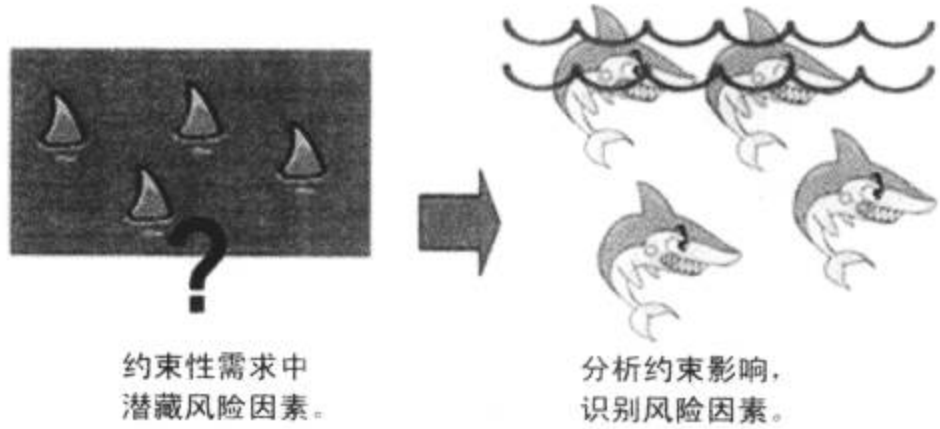


图 4-3 分析约束影响的隐喻

4.4 ADMEMS 方法的“约束分类理论”

那么，分析约束影响应当怎么做呢？

首先，我们要进行约束分类方式的革新，使它符合实践的需要。

总体而言，业界对约束性需求的重视和研究不够，而 ADMEMS 方法不仅注意到了约束对架构设计的重大影响，还强调约束分类理论应该直接体现“这些约束来自于哪些涉众”。如表 4-2 所示，4 类约束在 ADMEMS 矩阵中的位置清楚地表明：业务环境、使用环境、构建环境应分别考虑客户、用户、开发方这 3 类涉众，而技术环境则和 3 类涉众都有关。

表 4-2 4 类约束在 ADMEMS 矩阵中的位置

	广义功能	质量	约束
业务级需求	业务环境		技术环境
用户级需求		使用环境	
开发级需求		构建环境	

只有把握住涉众来源，才便于发现并归纳涵盖广泛的约束因素，也利于有针对性地进行交流，还可跟踪最终对约束的支持是否令涉众满意。

第一，来自客户或出资方的约束性需求。

架构师必须充分考虑客户对上线时间的要求、预算限制，以及集成需要等非功能需求。

客户所处的业务领域是什么？有什么业务规则和业务限制？

是否须要关注相应的法律法规、专利限制？

……

第二，来自用户的约束性需求。

软件将提供给何阶层用户？

用户的年龄段？使用偏好？

用户是否遍及多个国家？

使用期间的环境有电磁干扰、车船移动等因素吗？

……

第三，来自开发者和升级维护人员的约束性需求。

如果开发团队的技术水平有限（有些软件企业甚至希望通过招聘便宜的程序员来降低成本）、磨合程度不高、分布在不同城市，会有何影响？

开发管理方面、源代码保密方面，是否须要顾及？

……

第四，也不能遗忘，业界当前技术环境本身也是约束性需求。

技术平台、中间件、编程语言等的流行度、认同度、优缺点等。

技术发展的趋势如何？

……

架构师应当直接或（通过需求分析员）间接地了解和掌握上述需求和约束，并深刻理解它们对架构的影响，只有这样才能设计出合适的软件架构。例如，如果客户是一家小型超市，软件和硬件采购的预算都很有限，那么你就不宜采用依赖太多昂贵中间件的软件架构设计方案。

4.5 Big Picture：架构师应该这样理解约束

另外，还有一个重要的基础问题：太多架构师对约束的理解都过于零散，影响了系统化思维。为此，本节介绍架构师理解约束的“Big Picture”，如图 4-4 所示。

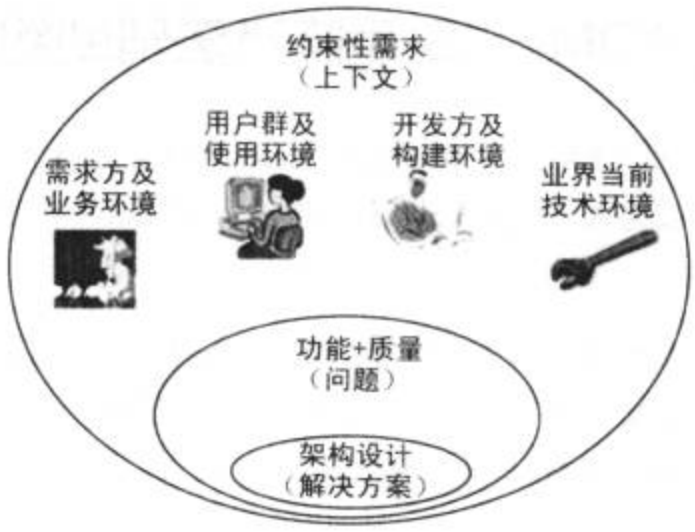


图 4-4　约束是架构设计要解决的问题的上下文，绝对不能忽视

一句话：约束是架构设计的上下文。

没有全局观念就不可能成为架构师，“约束是架构设计要解决的问题的上下文”是一个犀利的理解，揭示了“软件需求 = 功能需求 + 质量属性 + 约束”背后更深层次的规律。

如你所知，忽视了上下文对架构设计方案的限制，最终的架构设计就是不合理的，甚至是不可行的。举个生活当中的例子吧——设计大桥。如图 4-5 所示，建筑师必须关注以下 4 类约束的影响，合理规划大桥的设计方案：

- 考虑商业环境因素。以促进两岸城市间的经济交往为主（这会影响大桥的选址），同时决策层也希望大桥的建设在一定程度上起到提升城市形象的作用。
- 考虑使用环境因素。水上交通繁忙，而且常有大吨位船只通过，大桥建成投入使用期间不能对此造成影响。
- 考虑构建环境因素。这是一条很大的河流，水深江阔，为造桥而断流几个月是绝对不可行的。
- 考虑技术环境因素。斜拉桥因其跨度大等优点，当前广为流行，并且技术也相当成熟了……

图 4-5　设计大桥：也须考虑“4 类约束”的影响

4.6 用 ADMEMS 矩阵方法辅助约束分析

“PA-2 分析约束影响”在“PA-1 需求结构化”的基础上，充分考虑需求方及业务环境因素、用户群及使用环境因素、开发方及构建环境因素、业界当前技术环境因素等“4 类约束”，并分析约束影响、识别约束背后的衍生需求。

从本质上讲，分析约束影响就是分析各个需求项之间的关系，并发现被遗漏的需求。所以，将需求“化杂乱为清晰”的正交表可以作为分析约束影响的基础——即在需求项清晰定位的前提下，找到不同需求之间的关系、发现遗漏需求。

ADMEMS 矩阵方法应用法则有两个。

推导法则：从上到下，从右到左。

查漏法则：重点是质量属性遗漏。

第 4.7 节将通过案例予以说明。

4.7 大型 B2C 网站案例：需求结构化与分析约束影响

像 Amazon 这样大型的 B2C 网站，架构的起步阶段应如何规划呢？下面看 ADMEMS 方法的“表现”。

4.7.1 需求结构化

通过 ADMEMS 矩阵（需求层次—需求方面矩阵），有助于高屋建瓴地把握复杂系统的需求大局。如表 4-3 所示，先来梳理业务级的需求。

表 4-3 梳理业务级的需求

	功能	质量	约束
业务级需求	**业务目标及业务愿景** • 网站定位：B2C 零售 • 当前经营：图书 • 未来经营：图书、软件、音乐制品、电子产品、玩具、婴儿用品、化妆品、宠物、艺术品、杂货	**商业质量** • 新功能上线快，随需应变	**商业约束** • 投资 2000 万用于初期开发、运营、市场，之前须取得一定成功并融资成功 **集成约束** • 物流、银行、海关、实体店、各类提供商（包括工厂等生产企业，以及代理商等经销企业）
用户级需求			
开发级需求			

用户级需求，要特别注意挖掘来自“用户及使用环境”的约束。如表 4-4 所示，也勿忘开发方因素。

表 4-4 重点的用户级需求

	功能	质量	约束
业务级需求			
用户级需求	**用户** • 终端用户 • 各种员工角色	**运行期质量** • 易用性：最便捷的选择方式	**用户级约束** • 便捷的购物流程 • 客户群大：多国语言 • 客户群大：关注范围差异，须个性化 • 消费心理：营造集市效应，“别人也买了”、“别人还买了”
开发级需求			**开发方约束** • 新组建的团队

4.7.2 分析约束影响（推导法则应用）

接下来分析约束影响。

基于 ADMEMS 矩阵应用推导法则时规律十分明显：隐含需求（或遗漏需求）是通过从“上到下”或“从右到左”的脉络被发现的。如表 4-5 所示，考虑到公司的中远期发展（B2C 业务从图书扩展到各类商品），以及近期商业策略（投入资金 2000 万）的限制，必须制定“网站发展路线图”——而这对于架构而言属于“开发级约束”。

表 4-5 推导法则应用

	功能	质量	约束
业务级需求	**业务目标及业务愿景** • 网站定位：B2C 零售 • 当前经营：图书 • 未来经营：图书、软件、音乐制品、电子产品、玩具、婴儿用品、化妆品、宠物、艺术品、杂货	**商业质量** • 新功能上线快，随需应变	**商业约束** • 投资 2000 万用于初期开发、运营、市场，之前须取得一定成功并融资成功 **集成约束** • 物流、银行、海关、实体店、各类提供商（包括工厂等生产企业，以及代理商等经销企业）
用户级需求			
开发级需求			**开发方约束** • 网站发展路线图

表 4-6 和表 4-7 也是类似思维的体现。如此分析对架构师有一个额外的好处：理解了不同需求之间的联系，不再觉得需求是“一盘散沙”。

表 4-6 推导法则应用（续）

	功能	质量	约束
业务级需求	**业务目标及业务愿景** • 网站定位：B2C 零售 • 当前经营：图书 • 未来经营：图书、软件、音乐制品、电子产品、玩具、婴儿用品、化妆品、宠物、艺术品、杂货	**商业质量** • 新功能上线快，随需应变	**商业约束** • 投资 2000 万用于初期开发、运营、市场，之前须取得一定成功并融资成功 **集成约束** • 物流、银行、海关、实体店、各类提供商（包括工厂等生产企业，以及代理商等经销企业）
用户级需求	**用户** • 终端用户 • 各种员工角色 **管理员功能** • 灵活的打折设置 • 频率极高的新货上架		
开发级需求		**开发期质量** • 可扩展性	

表 4-7 推导法则应用（续）

	功能	质量	约束
业务级需求			
用户级需求	**用户** • 终端用户 • 各种员工角色 **终端用户功能** • 最快的全库搜索 • 评价功能（Web2.0） • 多角度关联信息 **管理员功能** • 灵活的打折设置 • 频率极高的新货上架		**用户级约束** • 便捷的购物流程 • 客户群大：多国语言 • 客户群大：关注范围差异，须个性化 • 消费心理：营造集市效应，“别人也买了”、“别人还买了”
开发级需求			

4.7.3 分析约束影响（查漏法则应用）

另外，还要主动运用查漏法则。例如，我们发现至今对质量的重视还不够（实践一线经常出现此情况），于是开始“查漏”：方方面面的约束背后藏着哪些必须强调的质量属性要求呢？如表

4-8 所示，如此多的集成要求，就必然要提高系统的互操作性……

表 4-8 还要主动运用查漏法则

	功能	质量	约束
业务级需求	业务目标、愿景 • 网站定位：B2C 零售 • 当前经营：图书 • 未来经营：……	商业质量 • 新功能上线快，随需应变	商业约束 • 投资 2000 万……
用户级需求	运行期质量 • 可伸缩性：几乎没有上限 • 性能：即强调速度，又强调吞吐量 • 安全性：数据安全 • 持续可用性：不停机		集成约束 • 物流、银行、海关、实体店、各类提供商（包括工厂等生产企业，以及代理商等经销企业）
	• 互操作性：含公司各系统间互操作		
开发级需求	开发期质量 • 可扩展性		

4.8 贯穿案例

4.8.1 PASS 系统背景介绍

PASS 系统的全称是：合理用药监测系统（Prescription Automatic Screening System）。通过全面部署 PASS 系统可以促进医院的用药合理化、规范化，降低医院的医疗事故率，还有利于管理部门高效全面地掌握医疗一线的用药情况，及早发现问题与解决问题。图 4-6 所示的信息有利于读者理解 PASS 系统的应用背景。

卫生部：二级以上医院2012年前设用药监测点

本报讯(记者 叶洲)卫生部近日发布了《加强全国合理用药监测工作方案》。方案指出，到2012年底，我国将全面运行覆盖全国二级以上医院的合理用药监测系统，完善药物合理使用和不良事件监测制度。

建成后的合理用药监测系统分为国家级监测系统和省级监测系统。国家级监测系统主要覆盖三级医院，以三级综合医院、中医医院为主；省级监测系统主要覆盖本辖区二级医院。

卫生部表示，通过收集医疗机构的用药相关医疗损害事件信息，一方面有利于相关部门及时掌握情况，迅速采取应对措施，降低对患者的损害及对社会造成的不良影响；另一方面通过多样本事件的分析，向医疗机构发布与用药相关医疗损害事件预警信息，有针对性地采取预防措施。

据了解，今年底将确定600家三级医院作为国家级监测点医院；2010年底，国家级监测点医院数量将扩大至900家三级医院；2012年底，各省级监测系统覆盖本辖区二级医院。

图 4-6 卫生部发布《加强全国合理用药监测工作方案》

（信息来源：*http://news.sina.com.cn/c/2009-02-08/002317168800.shtml*）

图 4-7 列出了 PASS 系统的主要功能。

- **医生**
  - -用药及时监测
  - -注意事项打印
- **管理员**
  - -身份管理
  - -用药规则数据库的更新
- **管理者（如院长）**
  - -多种方式的信息查询
  - -多张报表
- **外部系统的整合**
  - -药政部分的信息上报
  - -用药规则数据库的自动更新
  - ……

图 4-7 PASS 系统的主要功能

4.8.2 需求结构化

如你所知，将“功能列表”等同于“全部需求”根本不是架构师的应有做法。相反，为了全方位、多角度地把握需求，应当重视并运用“需求的结构”。表 4-9 为运用 ADMEMS 矩阵对 PASS 系统的需求进行结构化梳理的结果。

表 4-9 运用 ADMEMS 矩阵，对需求进行结构化梳理

	功能	质量	约束
业务级需求	**主管部门** • 统一监管 **医院** • 促进合理用药 • 降低医疗事故率	• 使用生命周期长 • 医疗行业的关键性	**业务环境约束** • 支持省级管理部门监管 • 药品繁多、用药规则繁多 • 用药规则随着时间会被修改
用户级需求	**用户** • 医生 • 院长 • 管理员		**使用环境约束** • 与很多医院早已部署的 HIS 系统协同工作 • 各医院 HIS 系统不统一，技术各异 • 分布式的使用要求 • 使用方便，避免孤立地使用 HIS 和 PASS 两套系统
开发级需求			**开发环境约束** • 人员水平不一

4.8.3 分析约束影响

下面逐一分析约束因素的潜在影响。

首先是业务级需求因素的影响分析，如表 4-10 所示。例如，既然药品的种类繁多、用药规则的数量也很大，就应该设法避免每家医院都重复录入用药规则——于是决定“省级集中提供用药规则的定义和更新支持”（这其实是业务流程一级的一项需求）。

表 4-10 分析约束影响：业务级需求因素的潜在影响分析

	功能	质量	约束
业务级需求	**主管部门** • 统一监管 **医院** • 促进合理用药 • 降低医疗事故率	• 使用生命周期长 • 医疗行业的关键性	**业务环境约束** • 支持省级管理部门监管 • 药品繁多、用药规则繁多 • 用药规则随着时间会被修改 • 省级集中提供用药规则服务以降低重复录入工作量
用户级需求	**用户** • 医生 • 院长 • 管理员	• 安全性 • 持续可用性 • 互操作（医院 PASS 和省级系统）	**使用环境约束** • 与很多医院早已部署的 HIS 系统协同工作 • 各医院 HIS 系统不统一，技术各异 • 分布式的使用要求 • 使用方便，避免孤立地使用 HIS 和 PASS 两套系统
开发级需求			**开发环境约束** • 人员水平不一

采用同样的思维方式，对用户群及使用环境一级的约束进行潜在影响的分析，如表 4-11 所示。例如作为产品的 PASS 系统要和很多医院的不同 HIS 系统整合在一起，就有可能重复开发 N 遍相同的程序部分，应通过提高“可重用性”来避免不必要的开销（否则维护起来也很不便）。

表 4-11 用户群及使用环境一级：分析约束影响

	功能	质量	约束
业务级需求	主管部门 • 统一监管 医院 • 促进合理用药 • 降低医疗事故率	• 使用生命周期长 • 医疗行业的关键性	业务环境约束 • 支持省级管理部门监管 • 药品繁多、用药规则繁多 • 用药规则随着时间会被修改 • 省级集中提供用药规则服务以降低重复录入工作量
用户级需求	用户 • 医生 • 院长 • 管理员	• 互操作（PASS 和 HIS 的整合） • 高性能 • 易用性	使用环境约束 • 与很多医院早已部署的 HIS 系统协同工作 • 各医院 HIS 系统不统一，技术各异 • 分布式的使用要求 • 使用方便，避免孤立地使用 HIS 和 PASS 两套系统
开发级需求		• 可重用性	开发环境约束 • 人员水平不一 • 降低重复开发的工作量

第5章
确定关键质量与关键功能

关键性的第一步是缩小范围……

——杰拉尔德·温伯格，《你的灯亮着吗》

事情简单勇者胜，事情复杂智者胜。面对时间压力和“复杂性怪兽”，我们有理由置疑“分析透彻所有需求，然后设计出架构”的做法。

——温昱，《软件架构设计》

如前所述，软件需求 = 功能需求 + 质量属性 + 约束。

第4章讨论了如何“分析约束影响”，本章将讲解 ADMEMS 方法 Pre-architecture 阶段的后两步：

- 第3步，确定关键质量。
- 第4步，确定关键功能。

5.1 为什么要确定架构的关键质量目标

人之所以痛苦，很多时候是因为追求错误的东西。

架构设计之初，要是制定了错误的质量属性目标（包括遗漏重要的质量属性），将面临的痛苦可能是客户不满、项目返工、同事抱怨……

在“需求结构化”的基础上，“确定关键质量”着重完成如下两项任务：

- 根据系统所在领域的特点及系统规模等因素，确定架构设计重点支持哪些质量属性（例如重点支持高性能、可扩展性……）。
- 分析上述质量属性之间的制约关系，第一时间指定权衡折衷的具体策略（例如明确高性能是第一位的，可扩展性与高性能相矛盾时应照顾高性能要求，是否引入支持可扩展性的设计须经架构组评审）。

形象的“挑物体”隐喻（如图 5-1 所示）可以帮助我们加深对本步骤工作的理解。

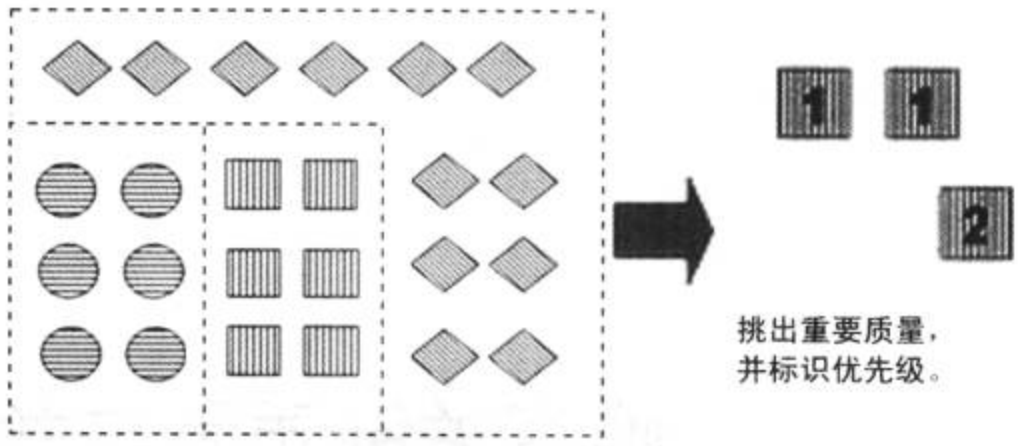


图 5-1　“确定关键质量”的隐喻

5.2　确定关键质量的 5 大原则

那么，如何确定关键质量呢？答案是，遵守和运用 5 大原则：

- 分类合适 + 必要扩充。
- 考虑多方涉众。
- 检查性思维。
- 识别矛盾 + 划定优先级。
- 严格程度符合领域与规模特点。

5.2.1　整体思路

确定关键质量的 5 大原则不是孤立的，也不是“甲乙丙丁，开中药铺”。图 5-2 概括了运用 5 大原则的整体思路。

质量的分类方式是基础，架构师对质量毫无研究绝对不行（图中 1）。

有了对不同质量属性的基础理解，就可以实施检查性思维了，基于质量进行分类看看，是否漏掉关键质量（图中 3）。

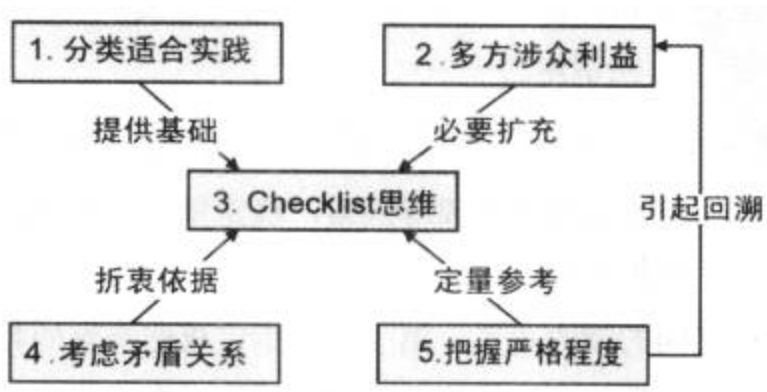


图 5-2　运用 5 大原则的整体思路

任何时候，只要关键质量属性多于一个，都要考虑它们之间是否有矛盾关系，在第一时间做出权衡取舍，确定不同的优先级（图中 4）。

那么，我们确定的关键质量属性子集是否合适呢？这要求架构师基于系统所处的领域、系统的规模大小来务实地判断（图中 5）。

如果当前确定的关键质量属性太少（例如设计软件平台时确定了 3、5 个关键质量），又怎么办呢？答案是，一定是架构师忽视了某些涉众的利益，须要进行“回溯”，找到遗漏的质量属性（图中 2）。

5.2.2 分类合适 + 必要扩充

有人说，有《ISO 9126》不就够了吗？答案是：不一定。

例如，《ISO 9126》定义的可靠性就远不及实际设计中的考虑全面（如图 5-3 所示）。可靠性包括成熟性、容错性、易恢复性等几方面，这是《ISO 9126》的定义；而有经验的架构师则更强调持续可用性（虽然在实践中两个词经常混淆）：

可靠性是指系统在规定时间内无故障工作的概率（可为 0~1 之间的所有值），它受“失效率”的影响。

持续可用性更严格，不仅失效（Failure）次数要尽量少，而且“因失效造成的中断的持续时间”也必须尽量短。

持续可用性包括了可靠性。

若是分布式系统，安全性差会随时造成系统瘫痪，持续可用性将大受影响，所以持续可用性也包括安全性高的隐含要求。

可维护性和可管理性也深刻影响着持续可用性。

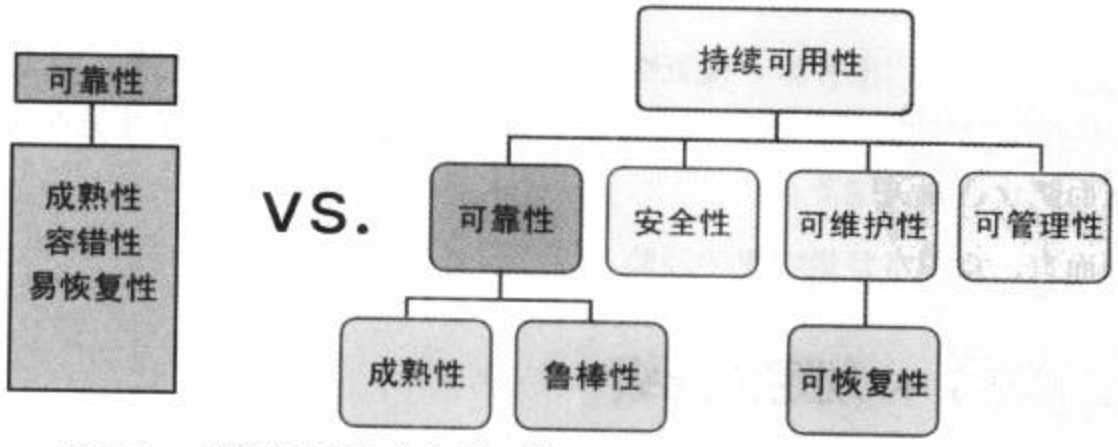


图 5-3 《ISO 9126》定义的可靠性就远不及实际设计中的考虑全面

由此可见，现存的标准更大程度上是对过去的总结，而架构师必须面对一些现实存在的新问题。所以，一线架构师应选择适合当前项目的分类方式（如 FURPS、ISO 9126、运行期质量 + 开发期质量等），并在必要时做一定扩充。

基于某种“质量分类方式”扩充一些新的质量属性要求，在实践中很常见。例如：

- 可管理性。系统一旦复杂，它本身的可管理性就非常重要。
- 开放性。既然平台等软件基础设施期望有较长的生命周期，那么开放性就相当重要，否则无法和其他技术整合。在实践中，开放性对可重用性有极大的影响。
- 仅有性能时补充效率。性能 ≠ 效率，例如用大型机支持一个简单的财务系统，性能高（用户笑了），效率差（老板哭了）。性能 = 速度 + 吞吐量 + 持续高速性。效率 = CPU、内存、硬盘和网络的利用率。
- 仅有效率时补充性能。同上。

5.2.3 考虑多方涉众

为什么会遗漏重要的质量属性呢？因为架构师没有全面地考虑多方涉众的利益。

首先，用户不仅需要功能，也需要质量。诸如性能、易用性等软件质量属性，并不像具体的软件功能那样直接帮助用户达到特定目标，但它们直接影响软件系统的成功与否。用户在使用软件系统的过程中，其关心的质量属性可能包括易用性、性能、可伸缩性、持续可用性、鲁棒性等。

有时，客户不一定是最终用户。例如，对超市销售系统而言，客户是某家连锁超市（的老板），而用户则是超市收银员和上货员。客户组织经常是分阶段、分部门进行信息化的，于是最终由此产生互操作性的质量要求。

架构师也应为开发人员而设计。例如，最关心可扩展性的人不是用户，而是乙方的开发维护人员。软件的可扩展性、可重用性、可移植性、易理解性、易测试性等也类似，它们都深刻影响着开发人员的工作，使开发更顺畅，抑或更艰难。

5.2.4 检查性思维

若能未雨绸缪，谁愿亡羊补牢？检查性思维这条原则颇为简单。说到底，这是一种意识，意识到“在架构设计之初”像“过一遍 Checklist”一样，看看每一项质量属性是否确实算不上“关键质量”，从而防止遗漏关键需求。

对架构师而言，意识常是最宝贵的经验。

5.2.5 识别矛盾 + 划定优先级

回顾“Conceptual Architecture 的故事”那一章中“都是 C++的错，换 C 重写”的故事，其问题的根源在于不懂得“质量属性之间常常存在矛盾关系”。类似的故事在你我身边并不鲜见。

所以，架构师必须有针对性地应对：在 Pre-architecture 阶段确定关键质量时考虑质量属性之间的矛盾关系。表 5-1 为质量属性关系矩阵图，任何一个“减号”都潜藏着风险。

表 5-1 质量属性关系矩阵

	性能	安全性	持续可用性	可互操作性	可靠性	鲁棒性	易用性	可测试性	可重用性	可维护性	可扩展性	可移植性	
性能				−	−	−	−	−		−	−	−	区域 1
安全性	−			−			−	−	−				
持续可用性					+	+							
可互操作性	−	−									+	+	
可靠性	−		+			+	+	+		+	+		区域 2
鲁棒性	−		+		+		+						
易用性	−					+		−					
可测试性	−		+		+		+			+	+		
可重用性	−	−		+	−			+		+	+	+	
可维护性	−		+		+			+			+		区域 3
可扩展性	−	−			+			+		+		+	
可移植性	−			+			−	+	+	−	+		

（“+”表示行促进列，“−”表示行影响列，“ ”表示行列两种质量属性之间影响不明显）

作为结果，架构师应确定关键质量的优先级，并在《架构文档》中明确记录此要求。例如，一个银行核心系统，你认为可重用性和安全性都很重要。此时，为了提高重用性，经常考虑直接引用第三方的 SDK，但这样做会降低安全性。最终，你作为架构师，若决定安全性的优先级高于可重用性的优先级，就会避免将某 SDK 用于关键传输数据的加密。

5.2.6 严格程度符合领域与规模特点

但是，不同系统的质量要求严格程度必然是迥异的，在确定关键质量时应如何把握呢？答案是：质量严格程度受到系统所处领域及系统规模的影响。

首先，不同领域对软件系统的质量要求不同。例如，航空航天、医疗设备控制软件等对可靠性、Safety 等都有严格要求，电子政务系统则要求信息的安全性……而基于 PC 的个人娱乐软件可接受的质量要求则宽松得多。

另外，就是规模。对同一领域的软件而言，产品的要求常常比项目高，平台的要求常常比产品高。为了理解这一点，图 5-4 给出了关键质量属性“个数”的常见经验值——进行定量说明。例如，作为一个银证转账产品或许可以不强调可管理性、可重用性、开放性等质量，但金融平台

的架构设计必须及早识别出这些关键质量。

经验值：项目：3-5项
产品：5-7项
平台：7-9项

例如
- 银证项目：
  易用性、安全性
- 银证产品：
  易用性、安全性、互操作性、
  可扩展性、可维护性
- 金融平台：
  安全性、互操作性、持续可用性、性能、
  可扩展性、可维护性、可重用性、可管理性、开放性

图 5-4　关键质量“个数”的定量经验值

5.3　为什么不是“全部功能作为驱动因素”

功能需求是大家最熟悉的一种需求。Karl E. Wiegers 在《软件需求（第2版）》中这样描述它：

功能需求（Functional Requirement）规定开发人员必须在产品中实现的软件功能，用户利用这些功能来完成任务，满足业务需求。功能需求有时也被称作行为需求（Behavioral Requirement），因为习惯上总是用“应该”对其进行描述：“系统应该发送电子邮件来通知用户已接受其预定”。

在第3章中，我们已经介绍了功能影响架构的具体原理：职责协作链。作为完整的软件系统，它在支持每一个具体功能时，都必然涉及软件不同“部分”之间的相互配合。系统的控制权在这些不同的“部分”之间来回传递，形成一条“职责协作链”……

再次仔细观察图 5-5，我们发现：完成不同功能的职责协作链之间可能有职责的重叠；在功能的数量比较多的时候，职责重叠将是不可避免的。

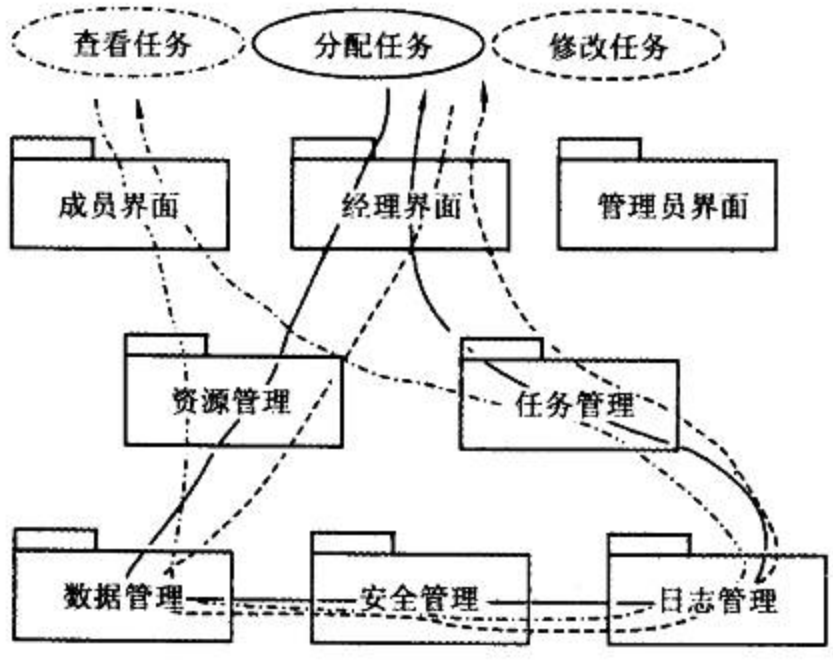


图 5-5　功能影响架构的基本原理：职责协作链

既然如此，以确定“将系统分为哪些部分、各部分之间如何交互”为中心的架构设计工作，就没有必要把“所有功能”都一视同仁地研究一遍。相反，如果能在所有功能中筛选出一个功能子集，研究功能子集中每个功能的职责协作链，从而识别出组成系统的所有职责单元（有经验的架构师还重视职责单元之间协作关系的到位体现），岂不更好？更何况，这样做有利于缓解“项目工期紧”的压力，并减小“需求变更”的冲击范围。

以需求结构化为基础，确定关键功能，将从“所有功能”当中筛选出对架构设计影响最大的一个“功能子集”（如图 5-6 所示）。

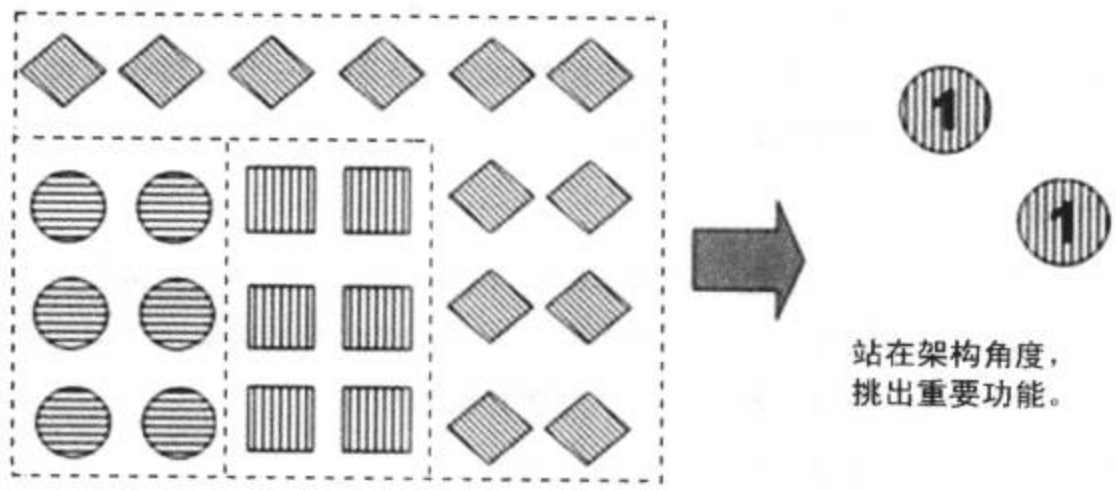


图 5-6 “确定关键功能”的隐喻

5.4 确定关键功能的 4 条规则

那么实践中如何做呢？

可通过如下 4 条启发规则，确定关键功能子集：

1. 核心功能。

2. 必做功能。

3. 高风险功能。

4. 独特功能（覆盖了上述 3 类功能没有涉及的职责）。

- 核心功能。

识别“核心功能”的标志是：业务层的接口要反映这些功能。例如，项目管理系统中，项目信息查看、添加项目任务等都是核心功能。

- 必做功能。

识别“必须实现的功能”主要依据客户方的背景。

有没有技巧呢？有。我们一直强调架构师不应忽视系统的《愿景与范围文档》，这份文档描述了项目立项的真正源起，文档“项目愿景的解决方案”中“主要特征”往往应作为“必做功能”的备选项。

另外，对于业务系统而言，一般支持“运营”的功能比支持“管理”的功能优先级要高。

- 高风险功能。

基于务实考虑，还应该把“风险高的功能”选入关键功能子集。

例如，你在设计一个网上书店系统，书籍的全库搜索功能就须要特别关注：

从用户角度讲，极慢的搜索速度，甚至直接收到“系统忙，请稍后再试”的提示，都是令人不满的；

从架构设计角度讲，此功能对书籍数据库进行“面状、只读”式的使用，与增加书籍、修改书籍信息等功能“点状、写入”式的数据库使用特点完全不同……尽早将全库搜索功能选入“高风险功能”之列，利于有针对性地进行架构设计。

- 独特功能。

最后，看看是否有覆盖了“上述3类功能没有涉及的职责”的功能。例如，如果你设计类似“搜狗拼音”这样的输入法软件，“词库在线更新“功能就必然是对架构关键的功能，因为忽略了它就很难发现架构中负责和服务器交互的“互操作模块”。

显然，“特殊功能”是相对上述3类功能而言的。

- 另外，架构师在确定关键功能子集时，还必须注意两点。

第一，“关键功能子集”的确定并不存在“标准答案”。

只要能较好地覆盖组成架构的不同职责模块，并体现职责模块之间协作关系的特点（有经验的架构师不会放过后者），那么“关键功能子集”的价值也就体现了。在为企业做架构内训时，曾有学员问：“能不能做到‘关键功能子集保证覆盖了所有职责模块’呢？”我笑了。首先，覆盖情况只有在系统基本开发完毕之后才能做“事后证明”，“事前证明”是不可能的。其次，为什么架构师的经验很重要呢？确定“关键需求”靠的就是经验，目的是用有限的时间，针对关键需求把设计做到位，并减小需求变更对架构设计的影响。

第二，值得专门说明的还有“关键功能所占比例”的问题。

有些专家认为比例大致是固定的，例如，Per Kroll 和 Philippe Kruchten 在《Rational 统一过程：实践者指南》中写道：

在初始阶段，应该确定出一些（大概占总数的20%～30%）对系统起关键作用的用例。这些用例通常对创建架构具有重要影响。

其实，“关键功能所占比例”不可能有一刀切的标准。例如，你要为一个只有5个功能的系统做架构设计（存在大量这样的系统：它们在计算、通讯、数据访问等方面有很高的复杂性，但功能的数量不多），选择20%的关键功能将意味着关键功能只有一个，这显然是不合适的。所以，“关键功能所占比例”应灵活确定：功能少的系统比例高些，功能多的系统比例少些。

5.5 大型 B2C 网站案例：确定关键质量与关键功能

接着第 4 章的 B2C 案例，继续运用 ADMEMS 方法对类似 Amazon 的大型网站进行分析。图 5-7 显示了架构设计驱动力的一部分：关键质量。对比第 4 章中此例的“分析约束影响”这一步（例如表 4-8），我们发现了一个重要现实：来自环境（或称上下文）的多方面的约束因素，对确定关键质量往往起着举足轻重的作用。

运行期质量：

- 可伸缩性：几乎没有上限
- 性能：即强调速度，又强调吞吐量
- 易用性:最便捷的选择方式
- 安全性：数据安全
- 持续可用性：不停机
- 互操作性：含公司各系统间互操作

开发期质量：

- 可扩展性

图 5-7 确定关键质量

下一步开始确定关键功能，如图 5-8 所示，大型 B2C 网站的核心功能必然包括检索、下订单、发货等反映核心业务流程的功能。

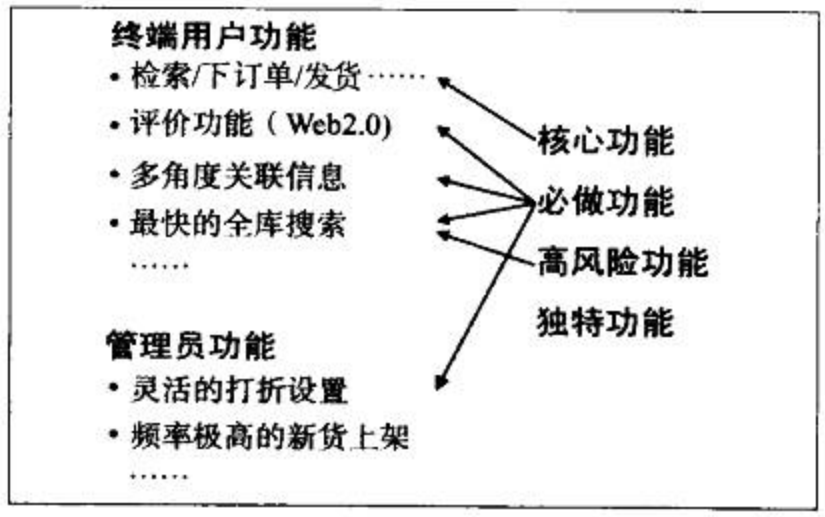


图 5-8 确定关键功能的中间思考结果：核心功能、必做功能、高风险功能

再找寻必做功能。Amazon 之所以是 Amazon，在于它的专业、便捷、实惠，而最终体现这些特点的功能就是必做功能：

评价功能（Web2.0）——专业。

多角度关联信息——专业。

最快的全库搜索——便捷。

频率极高的新货上架——便捷。

灵活的打折设置——实惠。

还要识别高风险功能，将之直接作为架构师深入关注的架构设计驱动力之一。“最快的全库

搜索”就是这样的功能，不仅在实现上有一定难度，还存在成本增加方面的压力。

最后识别独特功能。只有全面把握了整个系统覆盖的业务域（Business Area）或问题域（Problem Area），才能找出“覆盖了上述 3 类功能没有涉及的职责”的“独特功能”。图 5-9 展示了 B2C 典型的 6 个业务域。于是，架构师还必须重视的功能包括：信用卡支付、各种报表、补货管理、配送管理（亦展现在图 5-9 上）。

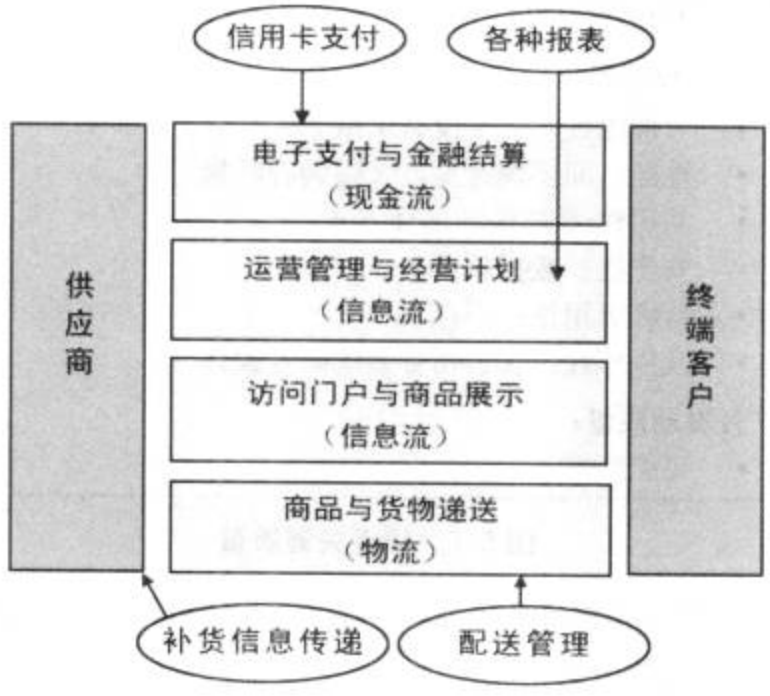


图 5-9　B2C 平台的 6 个关键业务域 与 特殊功能 的发现

5.6　贯穿案例

接下来考虑本书的贯穿案例——PASS 系统，确定对架构设计意义重大的质量需求和功能需求。如图 5-10 所示。对于一个用于医疗领域，部署于各家医院，须要和省级管理部门整合的 PASS 系统，安全性、高性能、易用性、持续可用性、互操作性都是架构设计须要特别关注的关键质量属性。对关键功能而言，无疑“检查处方”是核心功能；上级部门对各医院的用药监管是系统建设的基本目标之一，所以我们选择“上报用药信息”为必做功能；本系统没有风险特别高的功能；至此，再将“自动更新用药规则”以独特功能的“身份”选入关键功能之列，因为它覆盖了非常特殊且重要的“PASS 系统”和“用药规则信息中心系统”的互操作机制问题。

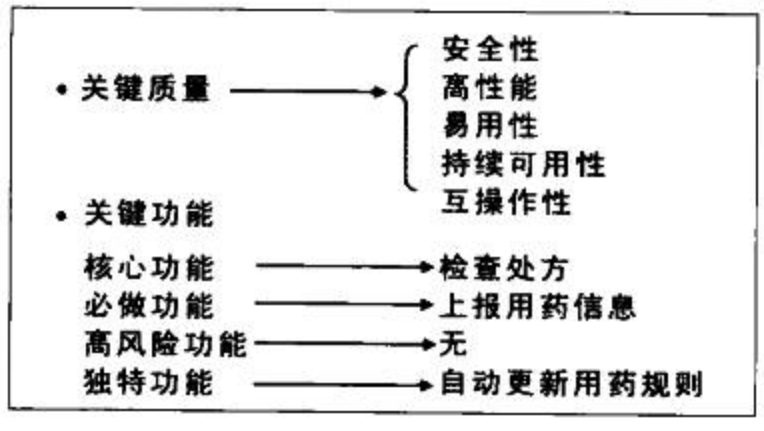
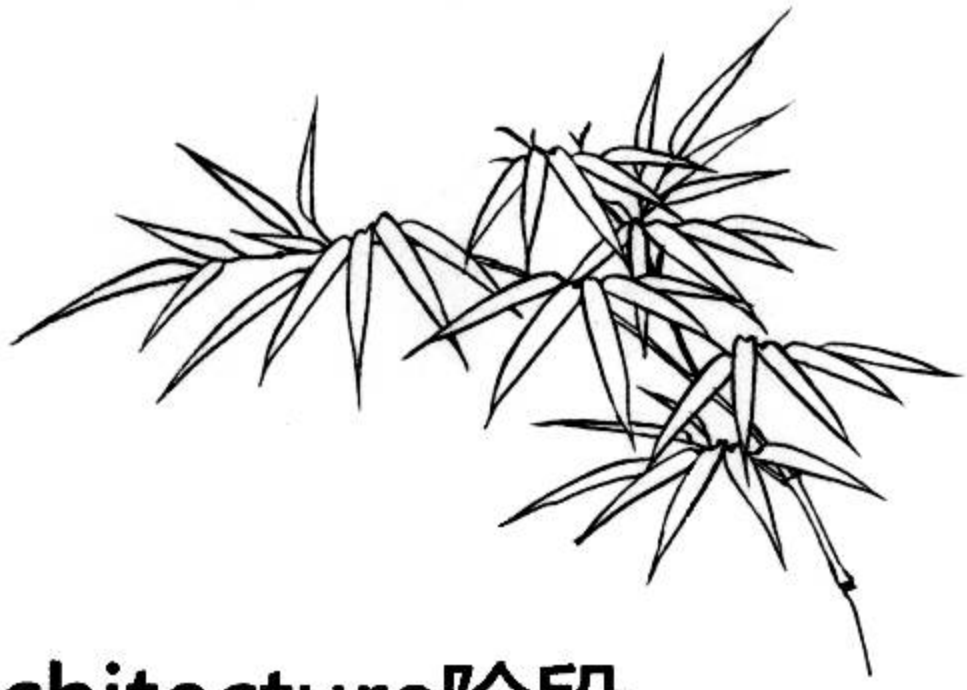


图 5-10　PASS 系统：确定关键质量和关键功能



Conceptual Architecture阶段

第6章
概念架构的故事

胜兵先胜而后求战，败兵先战而后求胜。

——孙子，《孙子兵法 · 形篇》

人们常常使用战术，而忽略了战略。战略要求从大局上把握整个架构与设计……架构错误的代价非常高。

——St é phane Faroult，《SQL 语言艺术》

架构新手和有经验架构师的区别之一，在于是否懂得，并能有效地进行概念架构设计。作为架构新手，尤其害怕碰上自己没做过的系统；系统较大时，一旦祭出“架构 = 模块 + 接口”的法宝却不太奏效，架构新手就往往乱了阵脚。相反，有经验的架构师不会一上来就关注如何定义“接口”，他们在大型系统架构设计的早期比较注重识别重大需求、特色需求、高风险需求，据此来设计概念架构。

另外，概念架构还是投标及售前工作的有力武器。金牌售前和普通售前的一个重要区别是，能否清晰地讲解概念架构，并借此说明“客户关心的价值如何实现、担心的问题如何解决”。

下面，通过两个发生在身边的故事，来一窥上述不同工作（架构设计、投标、售前）背后共同的幕后英雄——概念架构。

6.1 一筹莫展

日落西山，夜幕徐徐降临，遍布这个城市各个角落的写字楼陆续亮起了灯……

灯下，有我们这些软件从业者加班的身影。小张，还有老王，就是故事的主角。

6.1.1　小张，以及他负责的产品

加班人：小张

职业概况：28岁，某医疗软件公司的程序高手，这不，公司刚刚提拔他作了架构师。

加班缘由：他正负责一个名为“合理用药监测系统（Prescription Automatic Screening System，PASS）”的软件产品的架构设计。由于以前没有做过类似的产品，小张压力很大。按说，压力大对软件行业的人来说早已是家常便饭了，但要命的是，小张有些不知所措了……

图6-1所示的用例图列出了“合理用药监测系统（PASS）”的主要功能。

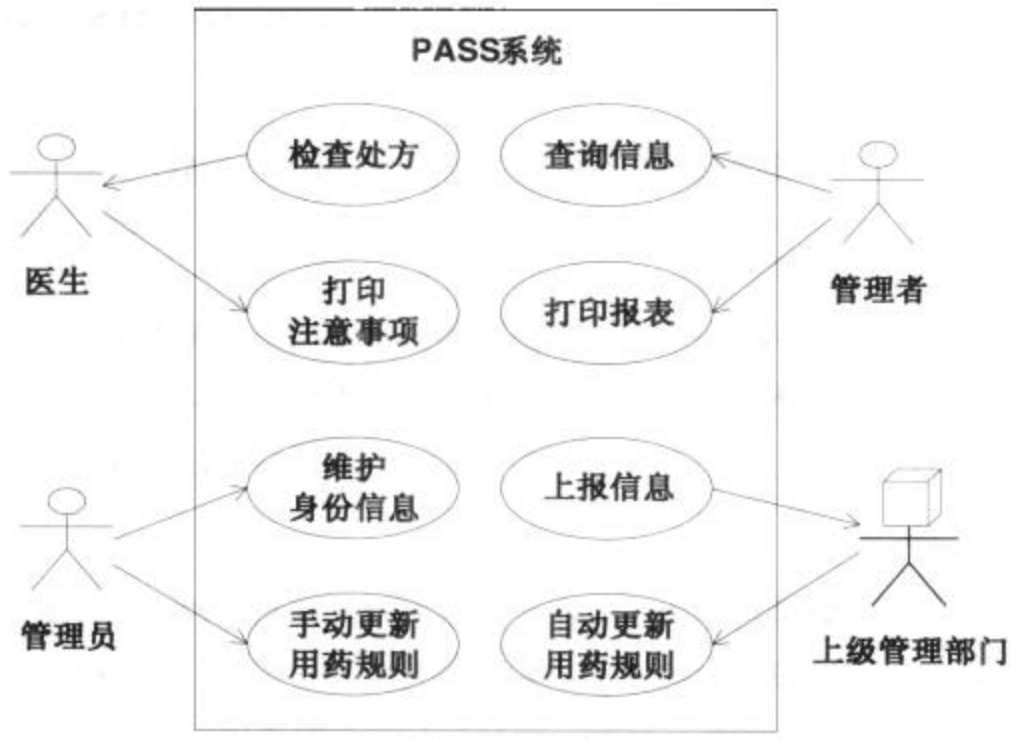


图6-1　PASS系统的用例图

晚7点，小张坐在桌前。

在心中，小张对架构的理解可以概括为一个公式：架构 = 模块 + 接口。

成为架构师伊始，他还专门用“接口”和“架构”作为关键词在网上搜了一把，看看别人的观点是否和他相同。结果让他非常满意，网上的一些观点和他的观点惊人地相似。例如，网上有观点认为：

“当你发现可以越来越灵活地使用接口时，那么你就从程序员升级为架构师了。

在一些大型项目或大型公司里，都是由架构师编写出系统接口，具体的实现类交给程序员编写。公司越大这种情况越明显，所以在这些公司里做开发，我们可能都不知道编写出的系统是个什么样子，每天的工作可能就是做‘填空题’了。”

但是，小张注定要在这个加班的夜晚，悄然开始重新认识“架构 = 模块 + 接口”这句话了。

一方面，小张已在“模块 + 接口”一级做了些设计努力。另一方面，小张也感觉到问题所

在了：这个 PASS 系统未来不可能仅仅包含一个可执行单元！相反，医生需要的功能要嵌入医院信息管理系统（HIS 系统）的医生工作站中，管理员的功能需要其他方式，Server 要独立出来……

小张当下的感觉，应了一句小品台词——“有点儿乱”。是呀，连 PASS 系统到底将包含几个“可执行单元”都没搞清楚，就考虑“模块 + 接口”一级的设计，的确有些武断了。

晚 9 点，思路不畅的小张开始上网搜资料。

在网上搜资料时，小张总是相当有耐心。他深知，虽然网上的资料非常多，但真正能启发思路的资料往往只在最后时刻出现。

小张移动鼠标，右击任务栏上的“IE 浏览器”按钮，“啪”地点击了“关闭组”菜单项。这表示小张认为查到的资料启发不大，准备重头来过——在今晚这是第 3 次了。

他若有所思，在新打开的 IE 浏览器中输入了 3 个搜索关键词：“架构”、“大局”、“不拘小节”。一篇博客文章引起了他的注意：

概念性架构就是对系统设计的最初构想，就是把最关键的设计要素和交互的机制确定下来，然后考虑具体技术的运用，设计出实际架构。

概念性架构应该抓大局、不拘小节。

虽然概念性架构都跳不出“架构 = 组件 + 交互”的基本定义，但它们描述架构的具体方式还是有比较大的差异：有的重视逻辑层，有的重视物理层，有的通过隐喻表明机制，有的看上去似乎就是一些设计元素的组合。不同的概念性架构图中，“连接”代表的含义千差万别：有的是依赖方向，有的是控制方向，有的是数据流向，因此，必须根据具体情况而定。

小张仔细地揣摩着每句话的意思。

不知不觉，时钟指向了 11 点，小张坐不住了。

他在办公室来来回回地踱步，表情时而郁闷，时而欣喜……

最后，小张把文章打印了一份，塞在包里，离开了办公室。

6.1.2 老王，后天见客户

加班人：老王

职业概况：35 岁，某电信软件企业网管软件事业部的售前工程师。老王从事软件行业有 10 年了，一直做软件开发，一年前开始做售前相关工作。

加班缘由：后天，他要到客户单位，做网管软件新产品的介绍。这个客户非常重要，而且公司对这一单志在必得，老王不敢怠慢。

老王看着公司草草拼凑出来的售前 PPT 有点儿发愁，架构方面的描述主要就是一个概念架构图，如图 6-2 所示。老王寻思，这种架构描述根本没有体现产品特点，叫做售前的如何说服客户呢？

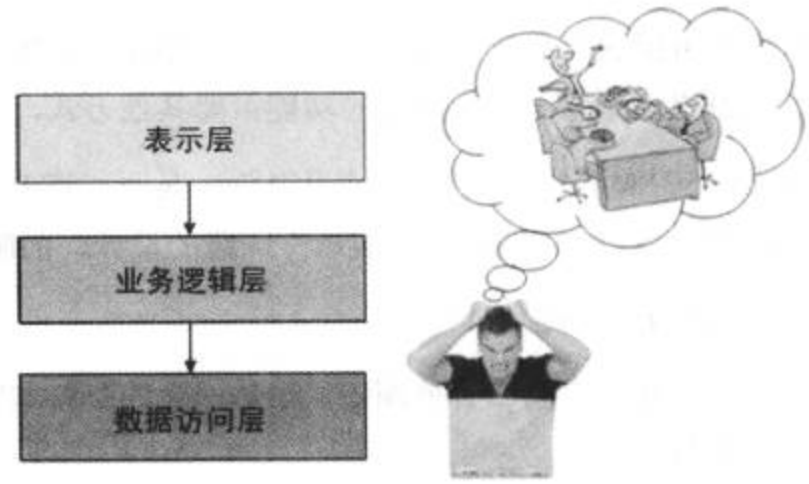


图 6-2　最初概念架构的描述没有体现产品特点

“我要是不说明，谁看了这个概念架构图能知道它是个网管系统，而不是电子商务或其他什么系统？”老王忿忿地想。

6.2　制定方针

小张，还有老王，昨天都加班到很晚，但今天他们依然按时来到单位。

每天的太阳都是新的。对开朗的人来说，昨天的烦恼算不了什么，根本不会影响他们今天工作的热情和创造力。

6.2.1　小张：我必须先进行概念架构的设计

小张在座位上坐定，破例没有查收邮件，更没有上网看新闻。他做的第一件事是，拿了一张干净的 A4 打印纸铺在桌上，开始逐条列举 PASS 系统需求中对架构产生最关键影响的“5 大因素”（如图 6-3 所示）。

1. 系统涉及不同的使用者。除了医生、管理员、院长之外，还包括“上级医疗主管部门系统”的使用者。
2. 分布式的特点明显。医生、管理员、院长不会在同一台机器上工作。
3. 集成的特点明显。和上级医疗主管部门的互操作要求，已经在需求文档中确定；而和医生考核系统等的互操作，也随时可能以“需求变更”的x形式出现。
4. 较高的持续可用性要求。PASS系统的责任就在于“防范于未然”，系统宕机将间接导致医疗事故率升高。
5. 降低HIS系统差异带来的影响。现在的HIS系统差异很大、实现技术不一，PASS的一部分要嵌入到HIS的医生工作站中，应在架构一级考虑重要策略以降低开发及维护成本。

图 6-3　PASS 系统：影响架构的“5 大因素”

“我作为架构师应该做些什么呢？”小张脑中快速地思考着，“是一成不变地继续‘模块 + 接口’一级的设计，还是先针对主要风险确定架构大局，而后再进一步考虑它呢？”

经过一系列反思，小张认为对 PASS 系统直接进行“模块 + 接口”一级设计存在以下两个严重的问题：

1. 针对“单独的可执行单元”形式的系统，或许可以直接按“模块 + 接口”方式展开架构设计，而现在的 PASS 系统显然不是。

2. 如何通过模块切分和接口定义来支持团队开发，还算不上当前的主要矛盾；上面分析的“5 大因素”才是当前的主要矛盾。

主要矛盾决定事态发展。想清楚了架构设计的主要影响因素，小张微微有些高兴。他认真地把他的决定写在笔记本上（还特意加了一句注解说明），生怕忘了似的：

> 我必须：首先根据对架构产生最关键影响的“5 大因素”进行概念架构的设计。
>
> 注：概念架构不关心明确的接口定义。

此时的小张俨然像个在战场上做出了重大战略决策的将军。

6.2.2 老王：清晰的概念架构，明确的价值体现

九点整，老王悠悠地走进办公室。他总是如此准时，在这个经常堵车的城市（笔者一次出差回来凌晨一点还堵车）可算得上一个小小的谜了。

他做的第一件事是，往茶杯里放入几十粒上好的枸杞，倒上开水。上午枸杞，下午绿茶，老王习惯了。老王就是有这本事，根本看不出来昨晚加班是多么地一筹莫展。老王或许不知道，这种闲庭信步的气质恰是他日后越来越成功的关键。

温伯格讲，“力量是一种关系”。老王很欣赏这句话。

音乐有力量吗？不一定！它可以催“人”泪下，但也可能是对“牛”弹琴。这里的关键不是音乐的良莠，而是音乐是否适合对象。

你的产品很好吗？不一定！只有满足客户需求的产品才是好的产品，所以做售前的决不能自我感觉良好。

胜利远远还没有“在望”，但端着茶杯愣了一会儿神儿的老王已成竹在胸：

> 后天的重点不是讲纯技术，而是抓住客户关心的价值和担心的问题，并在一个小时之内清晰地勾画出产品的相应策略。

6.3　柳暗花明

行必果。小张和老王忙活开了。

有人说，“行动果断是一种美德。”其实，他俩都觉得“行动果断”算不上什么美德，毕竟，老板是要看结果的——不行动，就永远没有成功的可能。

6.3.1　小张：重大需求塑造概念架构

如图 6-4 所示，小张将“分析需求特点”所归纳出的“5 大因素”作为概念架构设计的目标。

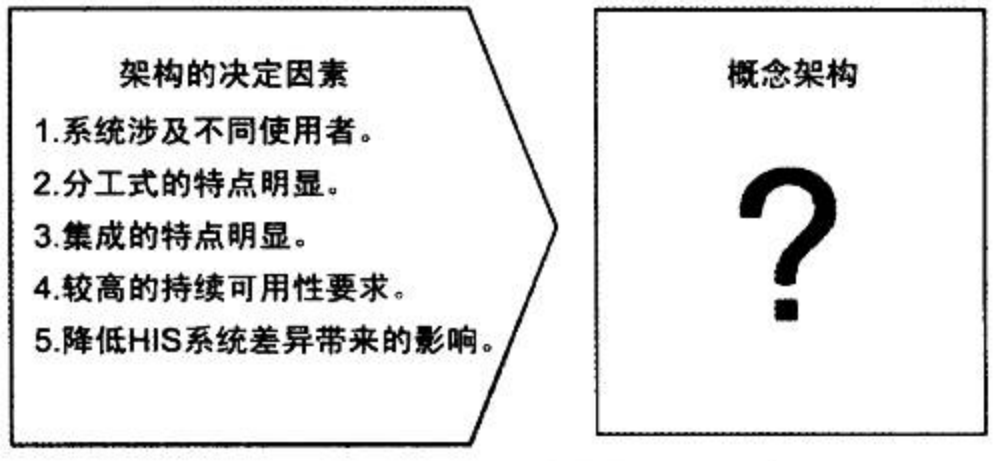


图 6-4　PASS 系统：概念架构的设计目标

考虑设计目标中的 1、2 和 3 这 3 点，小张得到了如图 6-5 所示的概念架构中间成果。

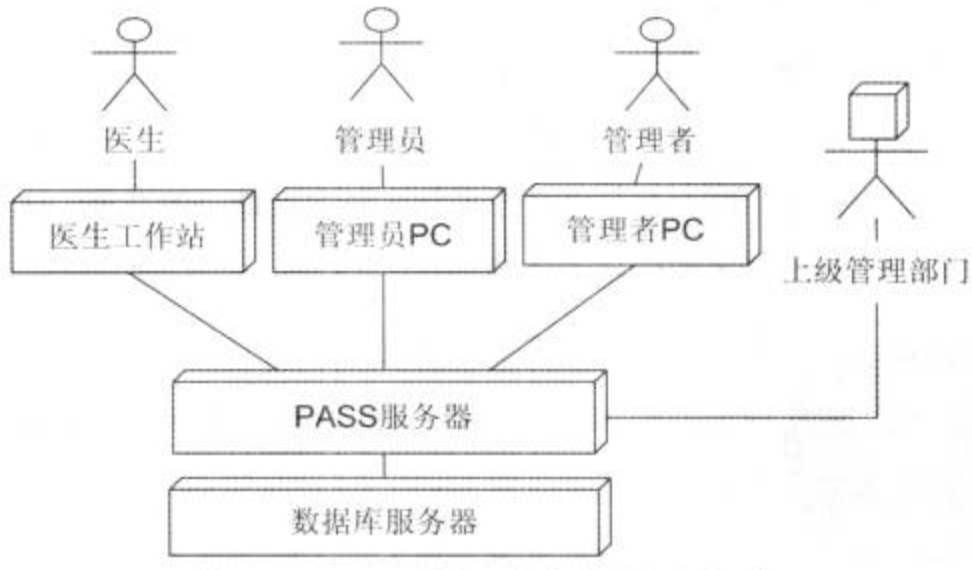


图 6-5　PASS 系统：概念架构的中间成果

接下来，小张继续深入概念架构的设计。他想：上述设计目标中的第 4 点和第 5 点还没有相应的对策，这无疑意味着巨大的风险，因为从现在的设计（如图 6-6 所示）来看根本无法做到“较高的持续可用性”，而对于“降低 HIS 系统差异带来的影响”也没有任何有针对性的设计决定。

经过一番考虑之后（具体思维过程请参考“目标—场景—决策表”方法的讲解），小张做出了如图 6-6 和图 6-7 所示的概念架构设计决定。为了提高持续可用性，在设计中引入了故障转移

群集，如图 6-6 所示。

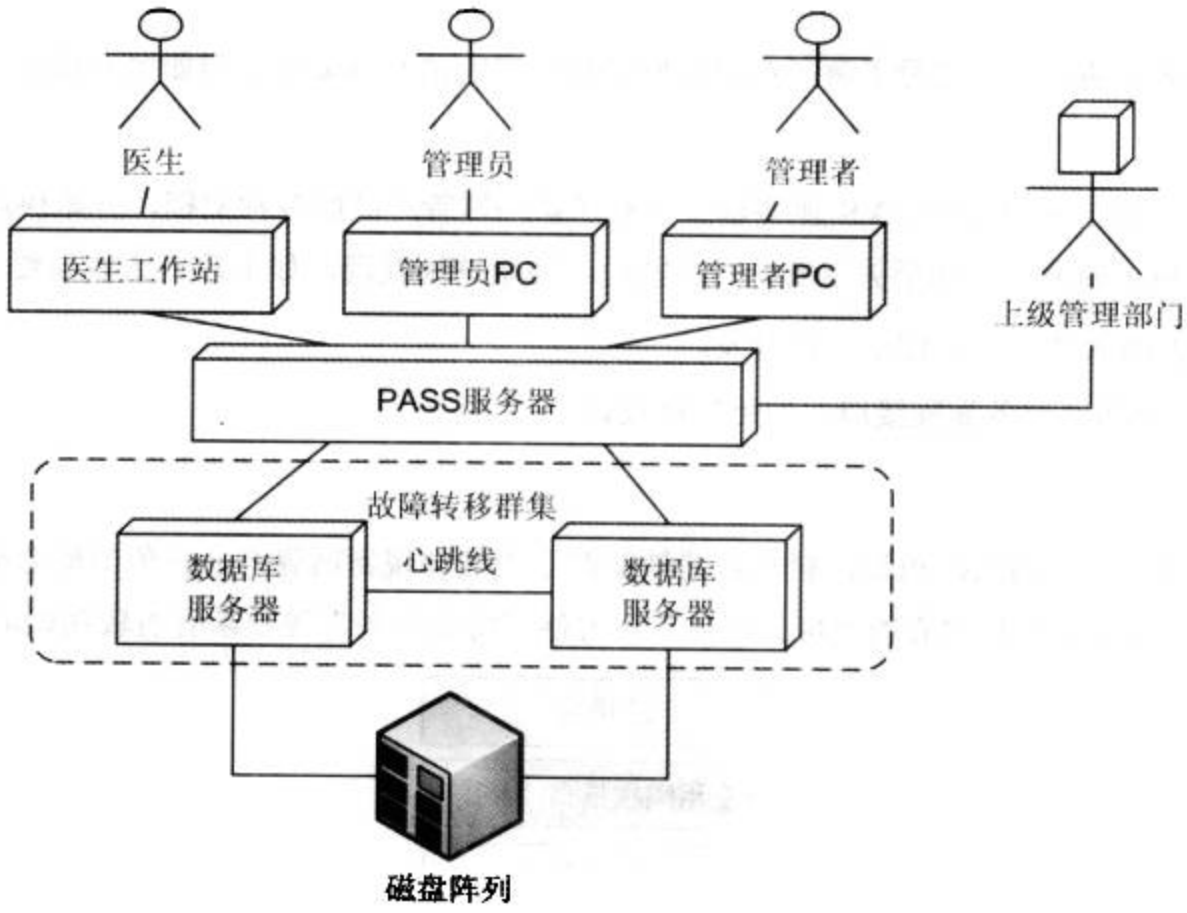


图 6-6 PASS 系统：增加了故障转移群集之后的概念架构

如图 6-7 所示的设计，主要针对“现行 HIS 系统差异很大、实现技术不统一”的约束性需求，重点考虑了如何提高重用性以降低开发及维护成本。采用的设计策略是：引入独立于具体 HIS 医生工作站的“PASS 系统医生模块通用 SDK”，它包括了“嵌入到医生工作站的软件模块”的所有特定 HIS 系统无关部分，使支持新的 HIS 的工作量降到最低。

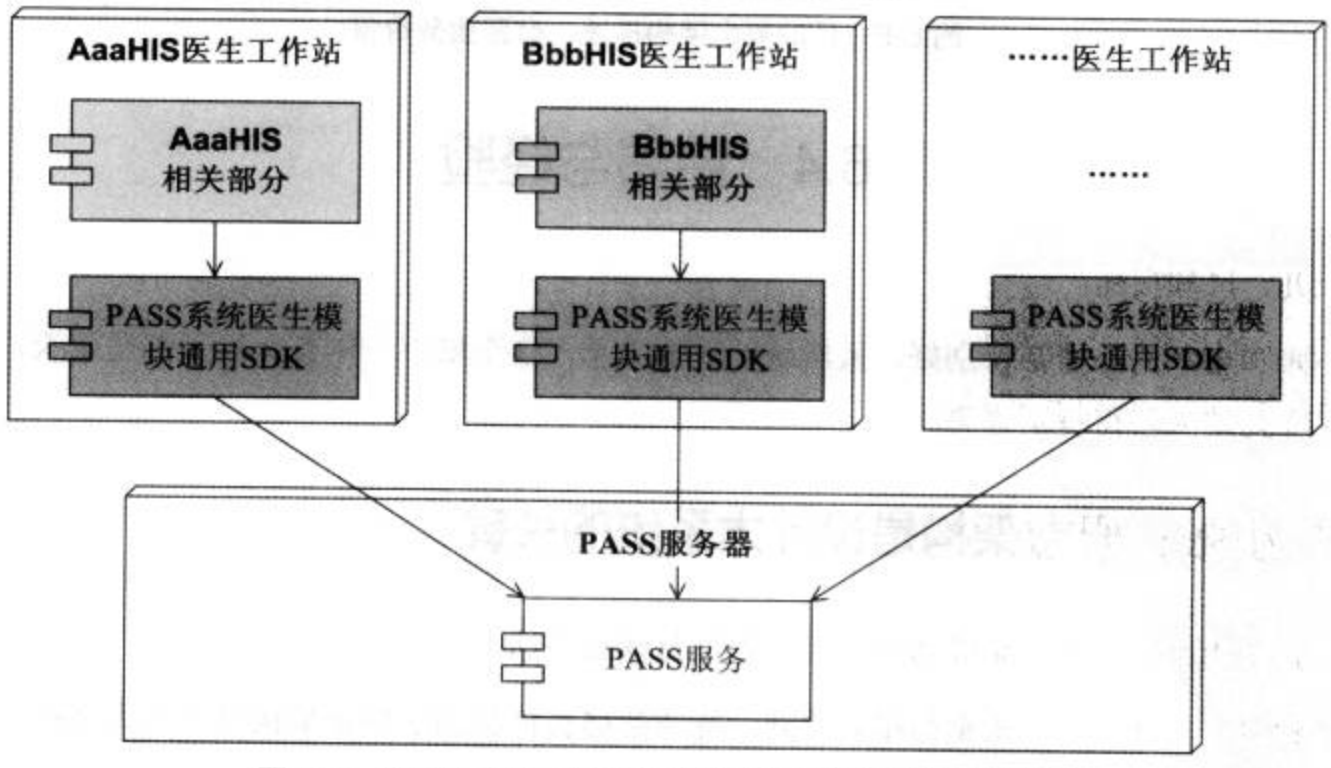


图 6-7 PASS 系统：概念架构中对 HIS 差异的应对策略

6.3.2　老王：概念架构体现重大需求

首先，老王通过一些途径了解了客户对网管软件采购的具体要求，例如可升级性、可方便支持新设备等。

接下来，老王从公司的 CVS 服务器上下载了新的网管产品的各种文档，开始快速浏览。他在找售前材料上遗漏的，却至关重要的产品特色。他敏感地发现，如下几点比较重要：

- 强大的 API 支持，便于二次开发。
- 基于 SPI（服务编程接口）的可扩展设计。

……

最终，老王运用 FAB 思维，轻车熟路地绘制了更能体现新网管产品价值的概念架构，如图 6-8 所示。此架构对客户关心的“可升级性、可方便支持新设备”等要求有着较明确的支持。

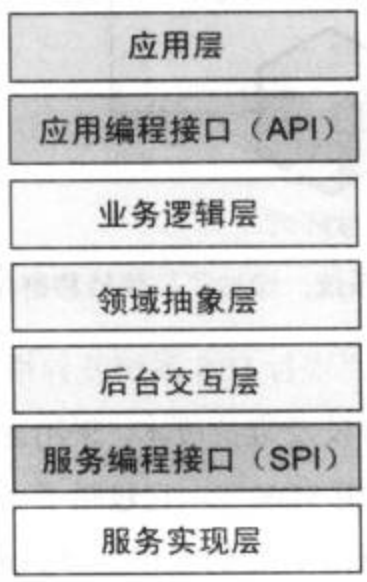


图 6-8　新的概念架构描述，凸显业务价值

6.4　结局与经验

天儿，风和日丽。

小张和老王的心情都特别好。虽然统计数据显示，这个城市一年有上百天都是蓝天，但他俩总觉得今儿的天蓝得特别漂亮。

6.4.1　小张：概念架构是设计大系统的关键

小张负责的 PASS 产品成功上市了，市场反映很不错。

回顾过往的辛苦，小张觉得很有收获，特别是切实体会到了概念架构设计对大系统成败的关键作用。他的工作笔记，也因而备受珍爱，其中一页写道：

万事开头难。

当要设计的软件系统非常复杂时，直接设计实际架构往往有困难。实际的软件架构设计过程是，一般应先进行概念架构的设计，把最关键的设计要素和交互机制确定下来。

概念架构是对系统设计的最初构想，但绝对不是无关紧要的；相反，它对大型系统的成功非常关键。架构师在设计概念架构时，必须牢牢抓住重大需求、特色需求、高风险需求，有针对性地确定设计策略。反过来讲，一个产品与类似产品在架构上的不同，其实在概念架构设计时就大局已定了。

概念架构一级的设计更重视“找对路子”，它往往是战略而不是战术，它比较策略化而未必全面，它比较强调重点机制的确定而不一定非常完整。

在概念架构设计中，不关注明确的接口定义；之后才是“模块 + 接口”一级的设计。对大型系统而言，这一点恰恰是必需的。

6.4.2 老王：概念架构是售前必修课

老王成功了，最终这家客户采购了老王所在的公司的新一代网管软件。

回顾这看似平常的一单，老张不无收获：

第一，概念架构是售前的必修课。所谓金牌售前，必备的能力之一是：能否清晰地讲解概念架构，并借此说明“客户关心的价值如何实现，担心的问题如何解决”。

第二，成功的售前必须关注客户。力量是一种关系，通过FAB思维找到产品之于客户的价值所在，是售前准备的重点之一。

第三，售前PPT不能千篇一律。作为公司，制作标准的售前PPT是为了避免一般售前人员不得要领，或者讲错理念，真正的专家级售前不应受到“标准售前 PPT”的限制。

第 7 章
Conceptual Architecture 总论

> “Use Case 驱动”的观点既有积极意义，也有不利影响。从积极的方面看，Use Case 这种需求描述方式确实有助于分析模型、设计模型、实现模型和测试模型的建立……但是从另一方面看，OOSE 对 Use Case 的依赖程度超出了它的实际能力。
>
> ——邵维忠，《面向对象的系统设计》

> 顶级设计者在设计中并不是按部就班地采用自顶向下（或自底向上）的方法，而是着眼于权重更大的目标。这些目标通常是难点问题，设计者不能轻易地看出这些问题的解决方案。为了得到整个问题的设计方案，设计者必须先致力于难点的设计并消除其中的疑惑。
>
> ——Robert L. Glass，《软件工程的事实与谬误》

概念架构是大型系统架构设计成败的关键。本章我们来熟悉概念架构的定义、实际意义、业界现状和实践要领。

7.1 什么是概念架构

图 7-1 所示是一座宏伟的大桥。这么复杂的架构，桥梁架构师是怎么“开始设计”的呢？

答案是：概念架构（Conceptual Architecture）。图 7-2 展示了斜拉桥的概念架构示意图。由此图可以看出，概念架构高屋建瓴地给出了高层解决方案：索塔负责承重，斜拉索吊起刚性梁。

图 7-1 一座复杂的大桥

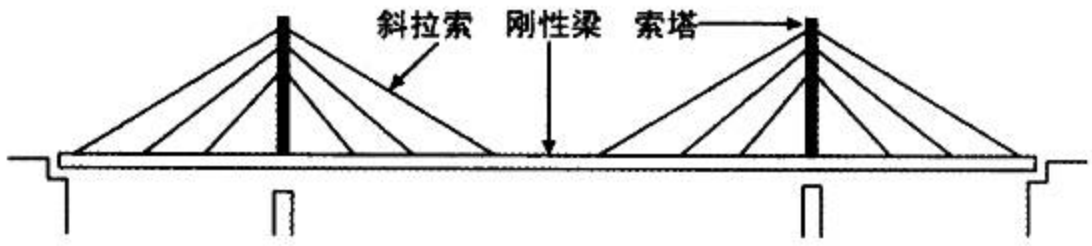


图 7-2 斜拉桥的概念架构示意图

下面，来看看软件行业（来自 Dana Bredemeyer 等专家）中概念架构的定义：

概念性架构界定系统的高层组件，以及它们之间的关系。概念性架构意在对系统进行适当分解，而不陷入细节。借此，可以与管理人员、市场人员、用户等非技术人员交流架构。概念性架构规定了每个组件的非正式规约及架构图，但不涉及接口细节。（The Conceptual Architecture identifies the high-level components of the system, and the relationships among them. Its purpose is to direct attention at an appropriate decomposition of the system without delving into details. Moreover, it provides a useful vehicle for communicating the architecture to non-technical audiences, such as management, marketing, and users. It consists of the Architecture Diagram (without interface detail) and an informal component specification for each component.）

根据定义，我们注意到如下几点：

- 概念架构满足“架构 = 组件 + 交互”的基本定义，只不过概念架构仅关注高层组件（high-level components）。
- 概念架构对高层组件的“职责”进行了笼统的界定（informal specification），并给出了高层组件之间的相互关系（Architecture Diagram）。
- 概念架构不应涉及接口细节（without interface detail）。

7.2 实际意义

不同的系统架构经常不同。但请继续追问自己两个极具价值的问题：

1. 不同系统的架构，为什么不同？

2. 架构设计中，应何时确立架构大方向的不同？

第 1 个问题的答案：需求不同，所以架构不同；当然，“需求”不是单指“功能需求”，而是包含了功能、质量、约束等方面。

第 2 个问题的答案：进行概念架构设计时应确立架构大方向。架构设计贵在有针对性，概念架构针对重大需求、特色需求、高风险需求的要求，给出高层次的解决方案——这就是概念架构最重要的意义。

另外，所谓“备选架构方案”经常是概念架构一级的，有助于架构的对比分析、评审优化。

最后，概念架构为投标、售前、市场宣传等工作提供强力支持，所以，概念架构也是售前和市场人员的“必修课”。

7.3 业界现状

7.3.1 误将“概念架构”等同于“理想架构”

主动思考以下两种说法是否正确：

1. 架构设计是功能需求驱动的，对吗？

2. 架构设计是用例驱动的，对吗？

说法 1，错误。因为，架构设计的驱动力 = 功能 + 质量 + 约束。

说法 2，同样错误。用例技术是功能需求实际上的标准，用例技术涉及，但无法全面涵盖非功能需求。所以，说法 2 和说法 1 其实并无本质区别。

因此，本书认为“用例驱动的架构设计”的做法颇值得商榷。纵观业界，有不少书持“用例驱动的架构设计”的观点，例如《Rational 统一过程：实践者指南》一书（本文献英文书名为《Rational Unified Process Made Easy: A Practitioner's Guide to the RUP》）。书中有一节名为“使用对架构重要的用例来驱动架构设计（Use Architecturally Significant Use Cases to Drive the Architecture）”，其中写道：

对架构重要的用例驱动了架构设计。对大多数系统而言，你通过选择仅仅 20%至 30%的用例，然后设计、实现并测试每个用例的一两个场景，就能降低大部分技术风险，并驱动架构的实现。为了实现某个特定用例，你要识别出那些支持用例的软件元素（Architecturally Significant Use Cases Drive the Architecture. For most systems, you can drive out a majority of technical risks and drive the implementation of the architecture by choosing the right 20 to 30 percent of use cases, and designing, implementing, and testing one or two scenarios for each use case. To implement a given use case, you need to identify which software elements are required to provide the functionality of that use case）。

实际上，也有实践者误认为概念架构就是只考虑功能而设计出来的理想化架构。其实，这是关于概念架构的最大误解，在实践中应当注意避免。

7.3.2 误把“阶段”当成“视图”

“视图”是架构领域的热门词汇，但很不幸，“视图是个筐，什么都往里装”——我们的同行常犯这种错误。例如，《编程匠艺——编写卓越的代码》一书错误地认为：

> 概念视图，有时也称为“逻辑视图”，这种视图显示了系统的主要部分及它们之间的相互关系。

再例如，一种称为“4 视图法”的架构设计方法在业界有一定影响，该方法也误把“概念架构阶段”当成了“概念架构视图”（如图 7-3 所示）。

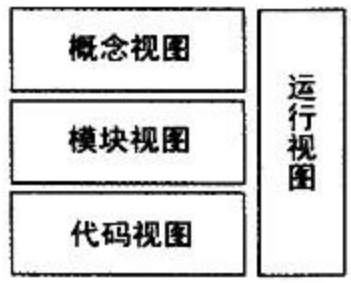


图 7-3 4 视图方法的错误：误把“概念架构阶段”当成了“概念架构视图”

（图片参考：《实用软件体系结构（Applied Software Architecture）》）

其实，视图与视图之间必须是并列的关系，是一种并行思维关系。概念架构不可能与“模块+ 接口”一级的设计并列，概念架构不是一个“架构设计视图”。正确的做法是，概念架构是一个“架构设计阶段”，必须在细化架构设计阶段之前，针对重大需求、特色需求、高风险需求，形成稳定的高层架构设计成果。阶段之于方法的意义和视图大不相同：

- 阶段是先后关系，视图是并列关系，这其中有本质区别。
- 不同阶段解决不同层次的问题——概念架构确定架构设计的大方针。
- 阶段应该与明确的里程碑相对应——概念架构确定的高层分割方案及其他重要决策是否合理？

7.4 实践要领

很多有经验的架构师已意识到了概念架构的重要性，却缺乏理性方法的指导。本节概述 ADMEMS 方法 Conceptual Arch 阶段的核心理念和 3 个步骤。

7.4.1 重大需求塑造概念架构

ADMEMS 方法 Conceptual Arch 阶段的核心理念：重大需求塑造概念架构，这里的“重大需

求”应涵盖功能需求、质量及约束 3 类需求中的关键部分。

为便于读者理解这一点的重要性，我们和“用例驱动的架构设计”做个对比，如图 7-4 所示。

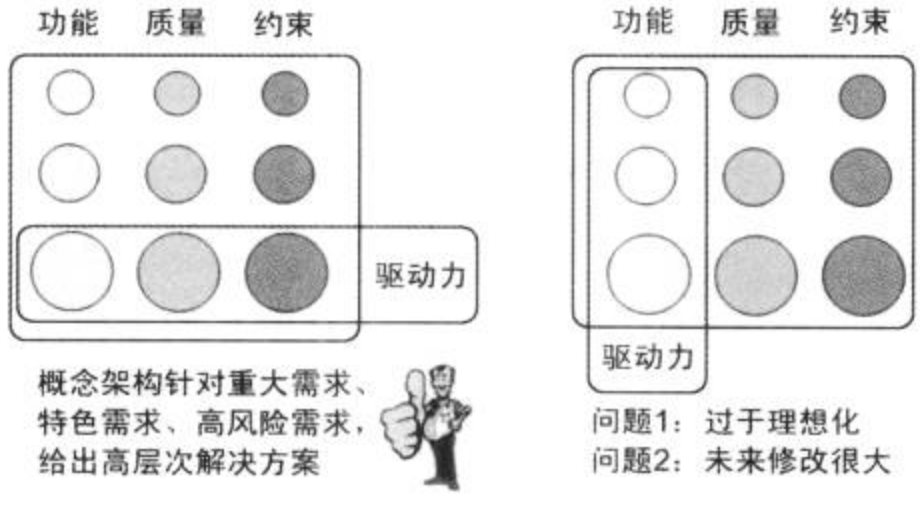


图 7-4 两种哲学的对比

显然，如果只考虑“功能需求”来设计概念架构，将导致概念架构沦为“理想化架构”，这个脆弱的架构不久就会面临“大改”的压力，甚至直接导致投标等工作失败。

7.4.2 概念架构阶段的 3 个步骤

大而言之，概念架构设计分为 3 个步骤（如图 7-5 所示）：

1. 初步设计。基于关键功能，借助鲁棒图进行以发现职责为目的的初步设计。这一步并不总是需要，但对于架构师而言，是“新系统”就必须重视这一步。

2. 高层分割。对系统这个黑盒子进行高层切分，例如切分复杂系统为多个二级系统，或者直接切分系统为具体子系统。

3. 考虑非功能需求。概念架构 ≠ 理想化架构，所以不仅要考虑功能，也必须考虑非功能。具体方法是采用 ADMEMS 推荐目标-场景-决策表。

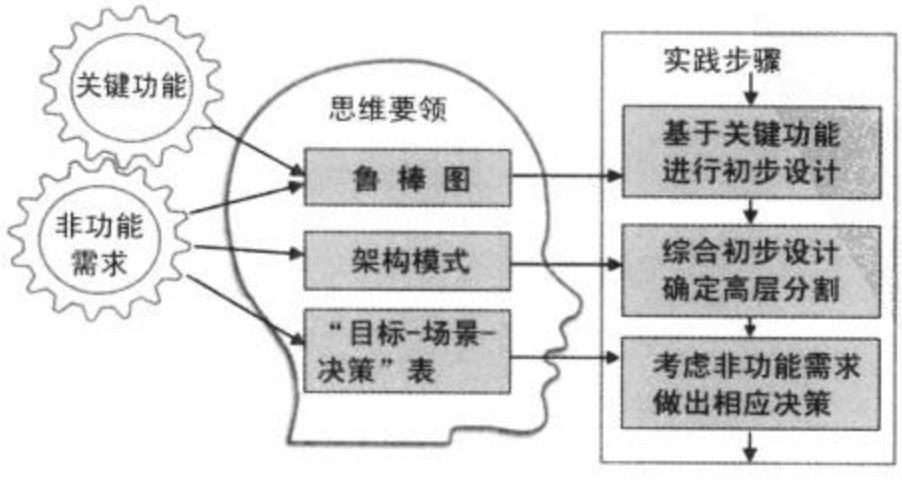


图 7-5 概念架构设计的具体步骤

第8章
初步设计

好的开始是成功的一半。

——谚语

所谓鲁棒性分析是这样一种方法：通过分析用例规约中的事件流，识别出实现用例规定的功能所需的主要对象及其职责，形成以职责模型为主的初步设计。

——温昱，《软件架构设计》

ADMEMS 方法的 Conceptual Architecture 阶段包含 3 个步骤：

- 第 1 步，初步设计。
- 第 2 步，高层分割。
- 第 3 步，考虑非功能需求。

本章讲解如何根据关键功能，借助鲁棒图进行初步设计。

8.1 初步设计对复杂系统的意义

初步设计并不总是必须的——架构师只有在设计复杂系统时才需要它。

另外，“复杂”与否还和“熟悉”程度有关。一个“很小”的系统涉及你未接触过的领域，你会觉得它挺复杂；一个“较大”的系统，但你有很具体的经验，你依然会觉得它“Just so so”。

初步设计的目标简单而明确：那就是发现职责。初步设计无须展开架构设计细节，否则就背上了“包袱”，这是复杂系统架构设计起步时的大忌。正如“初步设计”这个名字所暗示的，它只是狭义的架构设计的“第一枪”——之前的 Pre-architecture 阶段并未对“系统”做任何“切分”。

“初步设计”这个名字还暗示我们，后续的架构设计工作必然以之为基础。具体而言，初步设计识别出了职责，后续的高层分割方案才有依据，因为每个“高层分割单元”都是职责的承载体，而分割的目的也恰恰在于规划高层职责模型。

ADMEMS 方法强调“关键需求决定架构”的策略，“基于关键功能，进行初步设计”就是一个具体体现。如第 3 章 3.4.1 节所述，系统的每个功能都是由一条“职责协作链”完成的；而初步设计的具体思路正是“通过为功能规划职责协作链来发现职责”（如图 8-1 所示）。

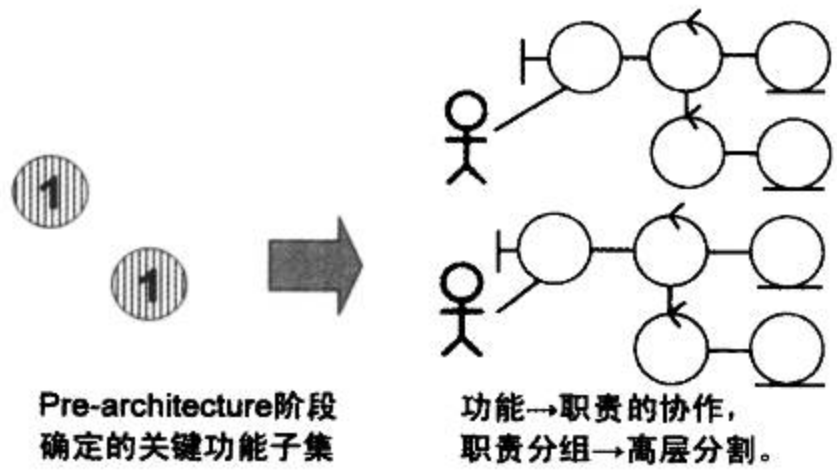


图 8-1　初步设计的具体思路：运用“职责协作链”原理

8.2　鲁棒图简介

ADMEMS 方法推荐以鲁棒图来辅助初步设计。那么，什么是鲁棒图呢？

8.2.1　鲁棒图的 3 种元素

鲁棒图包含 3 种元素（如图 8-2 所示），它们分别是边界对象、控制对象、实体对象：

- 边界对象对模拟外部环境和未来系统之间的交互进行建模。边界对象负责接收外部输入，处理内部内容的解释，并表达或传递相应的结果。
- 控制对象对行为进行封装，描述用例中事件流的控制行为。
- 实体对象对信息进行描述，它往往来自领域概念，和领域模型中的对象有良好的对应关系。

图 8-2　鲁棒图的元素

海象不是象，如此命名是因为“类比思维”在人的头脑中是根深蒂固的。关于鲁棒图 3 元素的“类比”，自然是 MVC。在图 8-3 中，我们做了更全面地对比，我们发现鲁棒图 3 元素和 MVC 还是有着不小的差异的。

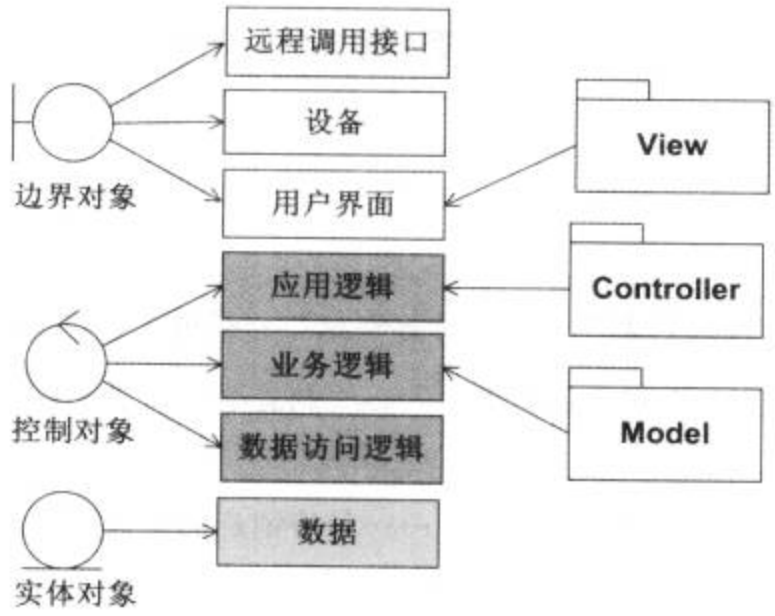


图 8-3 鲁棒图 3 元素和 MVC 的相似与不同

由图可以看出，鲁棒图 3 元素和 MVC 的主要不同在于：

- View 仅涵盖了“用户界面”元素的抽象，而鲁棒图的边界对象全面涵盖了三种交互，即本系统和外部“人”的交互、本系统和外部“系统”的交互、本系统和外部“设备”的交互。
- 数据访问逻辑是 Controller 吗？不是。控制对象广泛涵盖了应用逻辑、业务逻辑、数据访问逻辑的抽象，而 MVC 的 Controller 主要对应于应用逻辑。
- MVC 的 Model 对应于经典的业务逻辑部分，而鲁棒图的实体对象更像“数据”的代名词——用实体对象建模的数据既可以是持久化的，也可以仅存在于内存中，并不像有的实践者理解的那样直接就等同于持久化对象。

8.2.2 鲁棒图一例

图 8-4 展示的是银行储蓄系统的“销户”功能的鲁棒图。

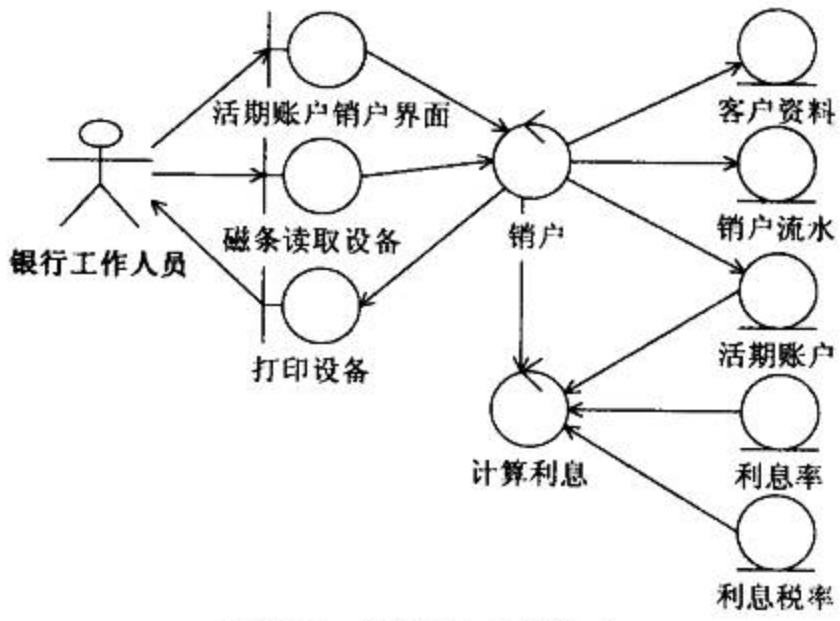


图 8-4 “销户”的鲁棒图

为了实现销户的功能，银行工作人员要访问3个“边界对象”：

- 活期账户销户界面。
- 磁条读取设备。
- 打印设备。

“销户”是一个“控制对象”，和“计算利息”一起进行销户功能的逻辑控制。

- 其中，“计算利息”对“活期账户”、“利息率”、“利息税率”这3个“实体对象”进行读取操作。
- 而“销户”负责读出“客户资料”……最终销户的完成意味着写入“活期账户”和“销户流水”信息。

8.2.3 历史

鲁棒图（Robustness Diagram）是由Ivar Jacobson于1991年发明的，用以回答“每个用例需要哪些对象”的问题。

后来的UML并没有将鲁棒图列入UML标准，而是作为UML版型（Stereotype）进行支持。

对于RUP、ICONIX等过程，鲁棒图都是重要的支撑技术。当然，这些过程反过来也促进了鲁棒图技术的传播。

8.2.4 为什么叫“鲁棒”图

也许你会问：“为什么叫‘鲁棒’图？它和‘鲁棒性’有什么关系？”

答案是：词汇相同，含义不同。

软件系统的“鲁棒性（Robustness）”也经常被翻译成“健壮性”，同时它和“容错性（Fault Tolerance）”含义相同。具体而言，鲁棒性指当如下情况发生时依然具有正确运行功能的能力：非法输入数据、软硬件单元出现故障、未预料到的操作情况。例如，若机器死机，“本字处理软件”下次启动应能恢复死机前5分钟的编辑内容。再例如，“本3D渲染引擎”遇到图形参数丢失的情况时，应能够以默认值方式呈现，从而将程序崩溃的危险度减小为渲染不正常的危险度。

而“鲁棒图（Robustness Diagram）”的作用有两点，除了初步设计之外，就是检查用例规约是否正确和完善。“鲁棒图”正是因为第2点作用而得名的——所以严格来讲“鲁棒图（Robustness Diagram）”所指的不是“鲁棒性（Robustness）”。

从Doug Rosenberg在《用例驱动的UML对象建模应用》的描述中，也可得到上述结论：

在ICONIX过程中，鲁棒分析扮演了多个必不可少的角色。通过鲁棒分析，您将改进用例文本和静态模型。

- 有助于确保用例文本的正确性，且没有指定不合理或不可能的系统行为（基于要使用的一组对象），从而提供了健康性检查（Sanity Check）。这种改进使用例文本的特性从纯粹的用户手册角度变为对象模型上下文中的使用描述。
- 有助于确保用例考虑到了所有必需的分支流程，从而提供了完整性和正确性检查。经验表明，为实现这种目标，并编写出遵循某些定义良好的指南的文本，而在绘制鲁棒图上花费的时间，将在绘制时序图时 3－4 倍地节省下来。
- 有利于发现对象，这一点很重要，因为在域建模期间肯定会遗漏一些对象。您还可以发现对象命名冲突的情况，从而避免进一步造成严重的问题。另外，鲁棒分析有利于确保我们在绘制时序图之前确定大部分实体类和边界类。

如前所述，它缩小了分析和详细设计之间的鸿沟，从而完成了初步设计。

8.2.5 定位

在本书中，关于鲁棒图最重要的一点是：它是初步设计技术。

不要再困惑于类似“鲁棒图是分析技术，还是设计技术”这样的问题了。大家只须记住两个公式：

- 需求分析 ≠ 系统分析
- 系统分析 ≈ 初步设计

关于“分析”与“设计”的区分，邵维忠教授和杨芙清院士在《面向对象的系统设计》一书中早已做过精彩阐释：

用“做什么”和“怎么做”来区分分析与设计，是从结构化方法沿袭过来的一种观点。但即使在结构化方法中这种说法也很勉强……

在“做什么”和“怎么做”的问题上为什么会出现上述矛盾？究其根源，在于人们对软件工程中“分析”这个术语的含义有着不同的理解——有时把它作为需求分析（Requirements Analysis）的简称，有时是指系统分析（Systems Analysis），有时则作为需求分析和系统分析的总称。

需求分析是软件工程学中的经典术语之一，名副其实的含义应是对用户需求进行分析，旨在产生一份明确、规范的需求定义。从这个意义上讲，“分析是解决做什么而不是解决怎么做”是无可挑剔的。

但迄今为止，在人们所提出的各种分析方法（包括结构化分析和面向对象分析）中，真正属于需求分析的内容所占的分量并不太大；更多的内容是给出一种系统建模方法（包括一种表示法和相应的建模过程指导），告诉分析员如何建立一个能够满足（由需求定义所描述的）用户需求的系统模型。分析员大量的工作是对系统的应用领域进行调查研究，并抽象地表示这个系统。确切地讲，这些工作应该叫做系统分析，而不是需求分析。它既是对“做什么”问题的进一步明确，也在相当

程度上涉及“怎么做”的问题。

忽略分析、需求分析和系统分析这些术语的不同含义，并在讨论中将它们随意替换，是造成上述矛盾的根源。

至于实践者为什么常将“需求分析”和“系统分析”混淆，这背后有着重要的现实原因。实际的工程化实践中，需求捕获、需求分析、系统分析不是完全孤立进行的。相反，它们往往是相互伴随、交叉进行的。需求工作伊始，无疑更多地是进行需求捕获工作，相伴进行的需求分析工作占的比例偏少；但随着掌握的需求信息越来越多，我们须要开展的对需求的分析和整理工作也越来越多了；而此时，伴随着对问题的分析，自然而然地会在高层次提出相应的应对策略……这，恰就是系统分析工作。《软件架构设计》一书中有如下阐述：

需求捕获是获取知识的过程，知识从无到有，从少到多。需求采集者必须理解用户所从事的工作，并且了解用户和客户希望软件系统在哪些方面帮助他们。

需求分析是挖掘和整理知识的过程，它在已掌握知识的基础上进行。毕竟，初步捕获到的需求信息往往处于不同层次，也有一些主观甚至不正确的信息。而经过必要的需求分析工作之后，需求更加系统、更加有条理、更加全面。

那么系统分析呢？如果说，需求分析致力于搞清楚软件系统要“做什么”的话，那么系统分析已经开始涉及“怎么做”的问题了。《系统分析》一书中写道：

简单地说，系统分析的意义如下：“系统分析是针对系统所要面临的问题，搜集相关的资料，以了解产生问题的原因所在，进而提出解决问题的方法与可行的逻辑方案，以满足系统的需求，实现预定的目标。”

需求捕获、需求分析，以及系统分析之间的关系我们必须理解透彻，否则会影响工作的有效进行。图 8-5 概括了三者之间的关系。

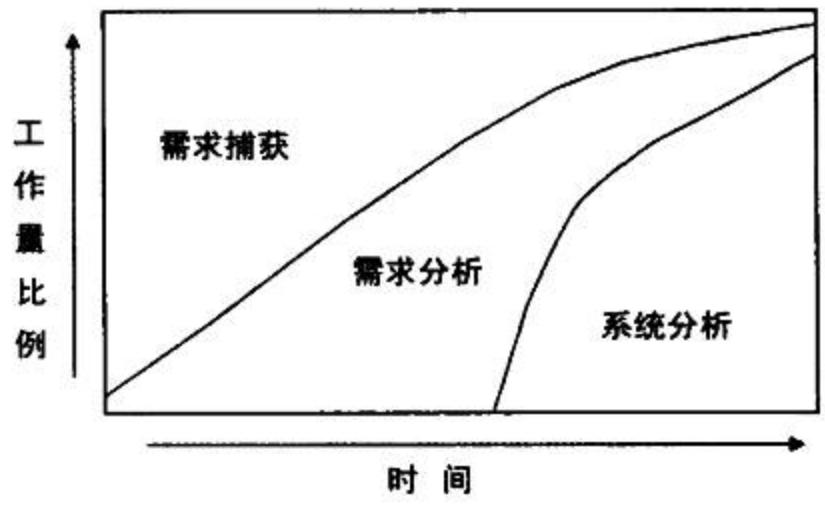


图 8-5　需求捕获、需求分析，以及系统分析之间的关系

再次强调，鲁棒图已经“打开”了“系统”这个“黑盒子”，将它划分成很多不同的职责，所以它是“设计技术”。

8.3 基于鲁棒图进行初步设计的 10 条经验

那么，如何借助鲁棒图进行初步设计呢？

ADMEMS 方法归纳了鲁棒图建模的 10 条经验要点（其中 50%是 ADMEMS 方法的原创经验，另一半来自业界其他专家），分别覆盖语法、思维、技巧、注意事项等 4 个方面（如图 8-6 所示），帮助一线架构师快速提升初步设计的能力。

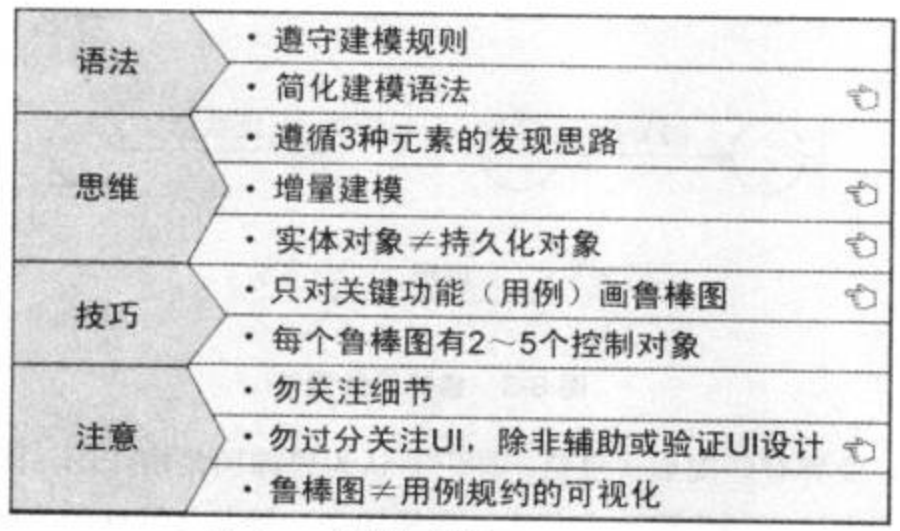


图 8-6 鲁棒图建模的 10 条经验

下面将逐一讲解鲁棒图建模的 10 条经验。

8.3.1 遵守建模规则

图 8-7 展示了鲁棒图的建模规则。Doug Rosenberg 在《UML 用例驱动对象建模》中写道：通过以下 4 条语句，可以理解该图的本质：

1） 参与者只能与边界对象交谈。

2） 边界对象只能与控制对象和参与者交谈。

3） 实体对象也只能与控制对象交谈。

4） 控制对象既能与边界对象交谈，也能与控制对象交谈，但不能与参与者交谈。

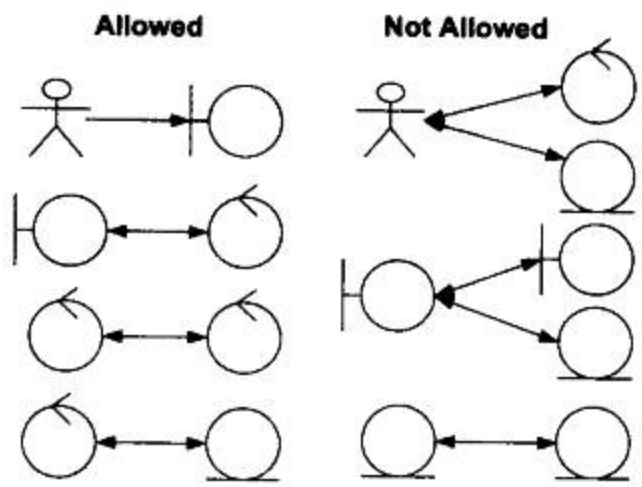


图 8-7 鲁棒图建模规则（图片来源：《UML 用例驱动对象建模》）

8.3.2　简化建模语法

图 8-8 展示了 ADMEMS 方法推荐的鲁棒图建模的语法。在实践中，简化的鲁棒图语法将有利于你集中精力进行初步设计，而不是关注细节——例如，鲁棒图根本不关心“IF 语句”怎么建模。

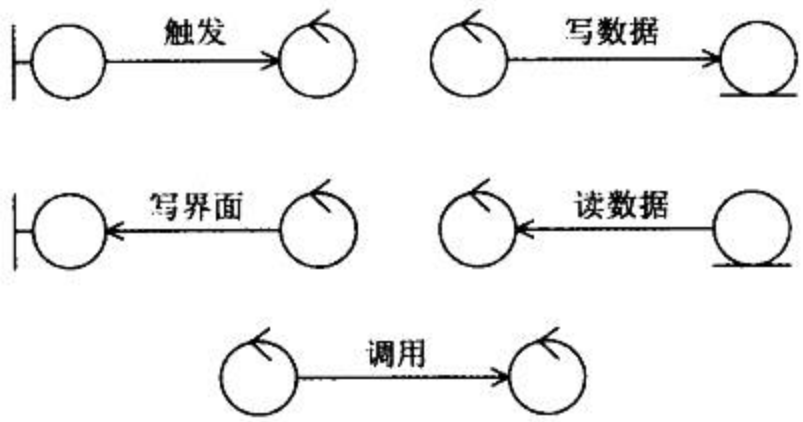


图 8-8　鲁棒图的语法

值得注意的是，业界有些观点（包括一些书）认为鲁棒图是协作图，因此造成了鲁棒图的语法非常复杂，不利于专注于初步设计。其实，鲁棒图是一种非常特殊的类图。

8.3.3　遵循 3 种元素的发现思路

图 8-9 说明了发现鲁棒图 3 种元素的思维方式。

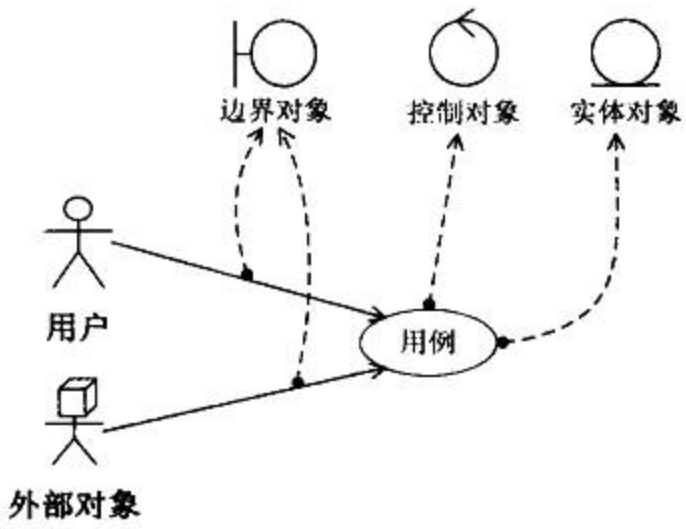


图 8-9　发现鲁棒图 3 种元素的思维方式

用例（Use Case）＝N 个场景（Scenario）。每个场景的实现都是一连串的职责进行协作的结果。所以，初步设计可以通过“研究用例执行的不同场景，发现场景背后应该有哪些不同的职责”来完成。

8.3.4　增量建模

“建模难”，有些人常如此感叹。例如，在画鲁棒图时，许多人一上来就卡在了“搞不清应

该有几个界面”的问题上，就会发出“建模难”的感叹。

下面演示“增量建模”这种技巧。从小处讲，增量建模能解决鲁棒图建模卡壳的问题；从大处讲，这种方式适用于所有种类的 UML 图建模实践。

例如，类似 WinZip、WinRar 这样的压缩工具大家都用过。请一起来为其中的“压缩”功能进行基于鲁棒图的初步设计。

首先，识别最明显的职责。对，就是你自己认为最明显的那几个职责——不要认为设计和建模有严格的标准答案。如图 8-10 所示，你认为压缩就是把原文件变成压缩包的处理过程，于是识别出了 3 个职责：

- 原文件。
- 压缩包。
- 压缩器（负责压缩处理）。

图 8-10 增量建模：先识别最“明显”的职责

接下来，开始考虑职责间的关系，并发现新职责。压缩器读取原文件，最终生成压缩包——嗯，这里可以将打包器独立出来，它是受了压缩器的委托而工作的。哦，还有字典……如图 8-11 所示。

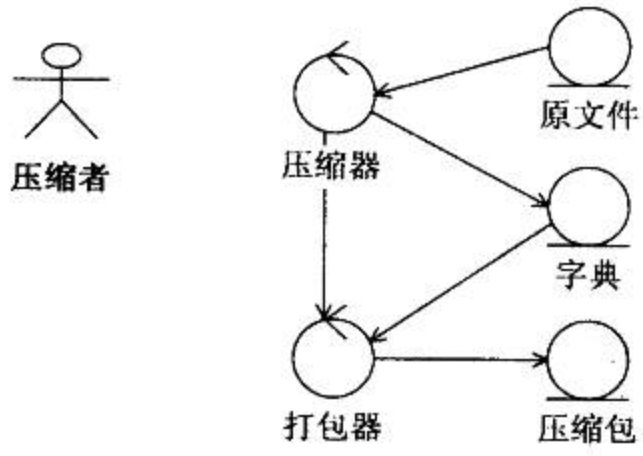


图 8-11 增量建模：开始考虑职责间的关系，并发现新职责

继续同样的思维方式（别忘了用例规约定义的各种场景是你的“输入”，而且，没有文档化的《用例规约》都没关系，你的头脑中有吗？）。图 8-12 的鲁棒图中间成果，又引入了压缩配置，它影响着压缩器的工作方式，例如加密压缩、分卷压缩或其他。

……最终的鲁棒图如图 8-13 所示。压缩功能还要支持显示压缩进度，以及随时取消进行了一半的压缩工作，所以，你又识别出了压缩行进界面和监听器等职责。

模型之于人，就像马匹之于人一样——它是工具。如果你不知怎样真正将“模型”为自己所用，反而被“建模”所累（经典的“人骑马、马骑人”的问题），请你问自己一个问题：

我是不是被太多的假设限制了思维？

或许，工具本身根本没有这样限制我！

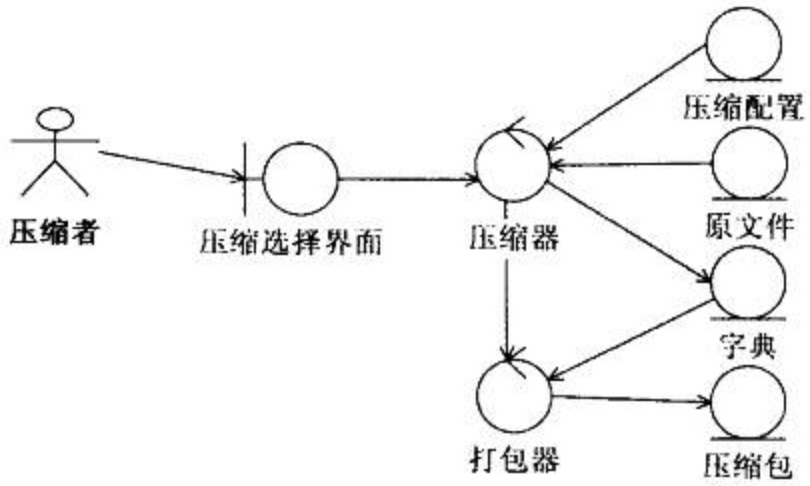


图 8-12　增量建模：继续考虑职责间的关系，并发现新职责

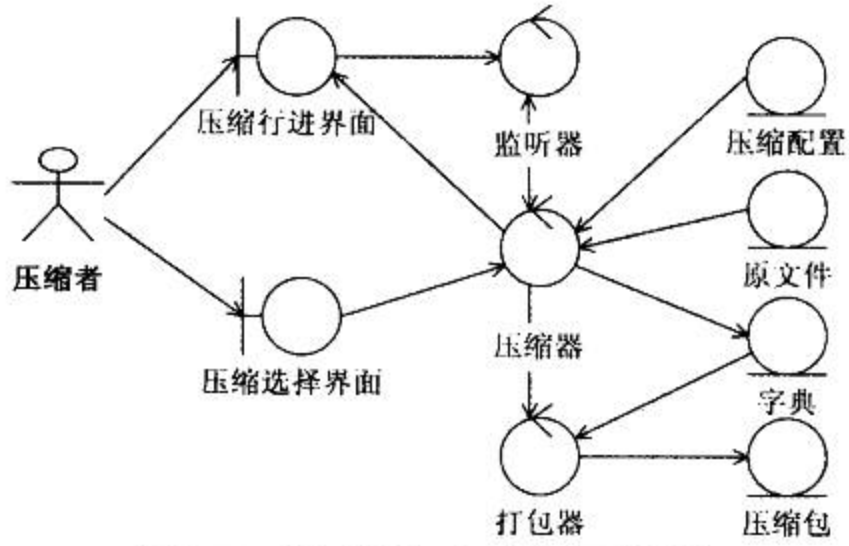


图 8-13　增量建模：直到模型比较完善

8.3.5　实体对象≠持久化对象

有的书上明确地说“实体对象就是持久化对象”，这是错误的。因为实体对象涵盖更广泛，它可以是持久化对象，也可以是内存中的任何对象。

一方面，在实践中，有些系统须要在内存中创建数据的“暂存体”以保持中间状态，这当然可以被建模成实体对象。另一方面，有的系统没有持久化数据，但基于鲁棒图的初步设计依然可用，此时难道鲁棒图不包含实体对象？显然不对。

因此，实体对象≠持久化对象，这个正确认识将有助于你的实践。

8.3.6 只对关键功能（用例）画鲁棒图

基于“关键需求决定架构”的理念，功能需求作为需求的一种类型，在设计架构时不必针对每个功能都画出鲁棒图。

8.3.7 每个鲁棒图有 2～5 个控制对象

既然是初步设计，鲁棒图建模时，针对关键功能的每个鲁棒图中的控制对象不必太多太细，5 个是常见的上限值。

相反，若实现某功能的鲁棒图中只含 1 个控制对象，则是明显地“设计不足”——这个控制对象的名字必然和功能的名字相同，这意味着没有对职责进行真正的切分。例如，WinZip 的压缩功能设计成图 8-14 所示的鲁棒图，几乎没有任何意义。

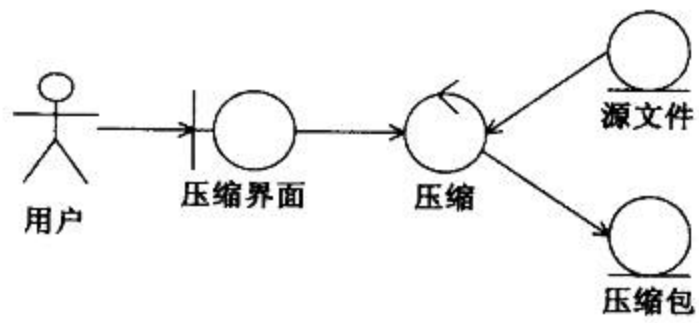


图 8-14 只有一个控制对象的鲁棒图明显“设计不足”

8.3.8 勿关注细节

初步设计不应关注细节。例如，回顾前面图 8-4 所示的“销户”的鲁棒图：

- 对每个对象只标识对象名，都未识别其属性和方法。
- “活期账户销户界面”，具体可能是对话框、Web 页面、字符终端界面，但鲁棒图中没有关心这些细节问题。
- “客户资料”等实体对象须要持久化吗？不关心，更不关心用 Table 还是用 File 或其他方式持久化。
- 没有标识控制流的严格顺序。

8.3.9 勿过分关注 UI，除非辅助或验证 UI 设计

过分关心 UI，会陷入诸如有几个窗口，是不是有一个专门的结果显示页面等诸多细节之中，初步设计就没法做了。

别忘了，初步设计的目标是发现职责。初步设计无须展开架构设计细节，否则就背上了“包袱”，这是复杂系统架构设计起步时的大忌。

8.3.10 鲁棒图≠用例规约的可视化

鲁棒图是设计，“系统”已经被切分成不同的职责单元。而用例规约是需求，其中出现的“系统”必定是黑盒（如图 8-15 所示）。所以，两者有本质区别。

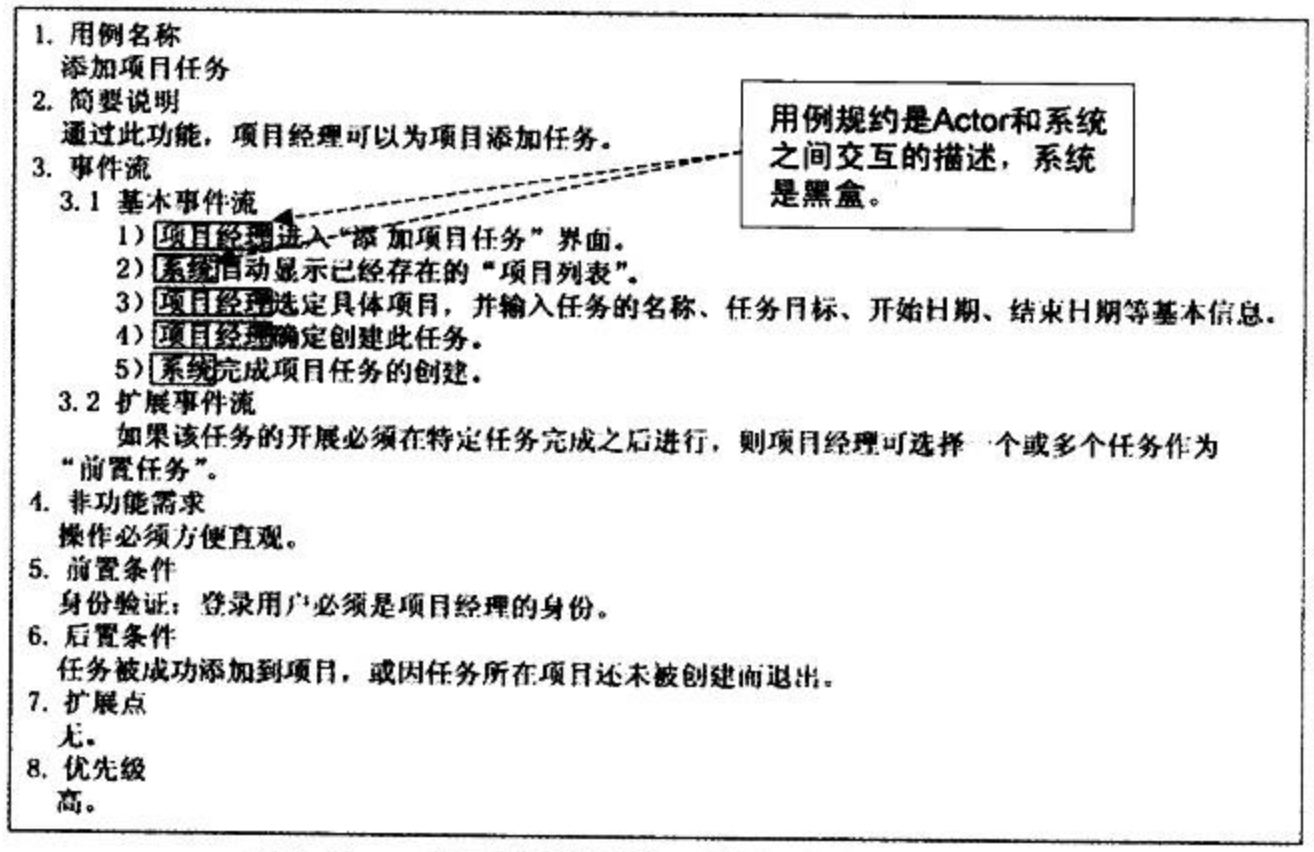


图 8-15 一个用例规约示例：需求意味着系统保持黑盒

8.4 贯穿案例

接下来考虑本书的贯穿案例 PASS 系统，如何借助鲁棒图进行初步设计呢？

再次明确以下几点：

- 初步设计的目标是发现职责，为高层切分奠定基础。
- 初步设计不是必须的，但当待设计系统对架构师而言并无太多直接经验时，则强烈建议进行初步设计。
- 基于关键功能（而不是对所有功能），借助鲁棒图（而不是序列图）进行初步设计。

下面，一起思考如何针对“实时检查处方”功能进行初步设计——重点体会“增量建模”的自然和强大。

首先，识别最明显的职责。如图 8-9 所示，先识别出了最不可或缺的、体现整个功能价值所在的与“处方检查结果”相关的几个职责。

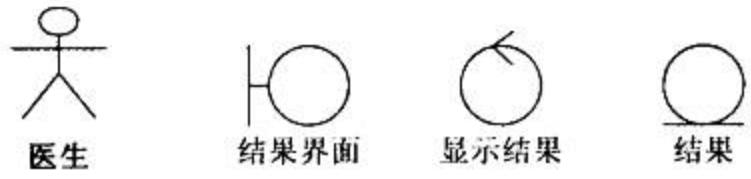


图 8-16 实时检查处方：先识别最明显的职责

接下来开始考虑职责间的关系，并发现新职责。检查结果是如何产生的呢？检查这个控制对象，读取处方和用药规则信息，最终生成了处方检查结果。如图 8-17 所示。

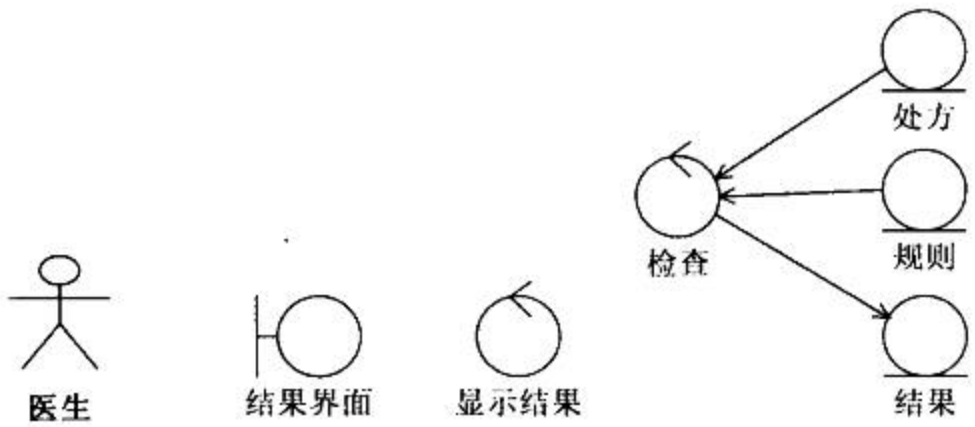


图 8-17 实时检查处方：开始考虑职责间的关系，并发现新职责

OK，如此一来，解决了“结果是怎么来的”这个问题。如图 8-18 所示。

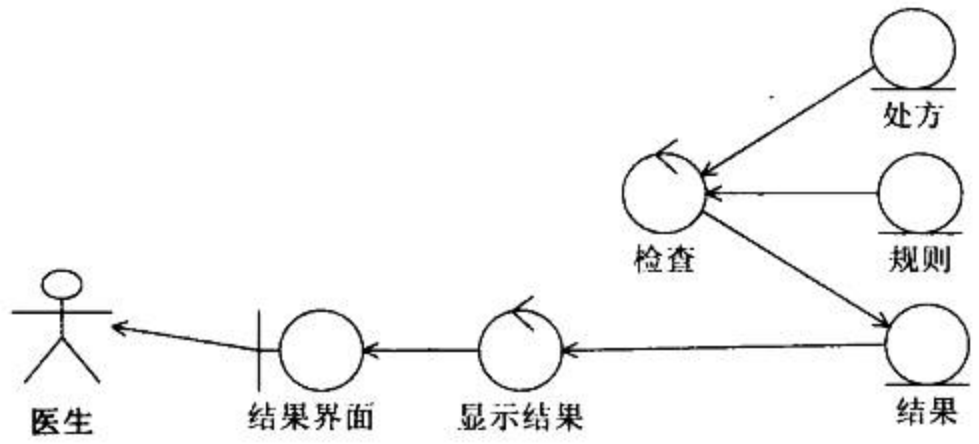


图 8-18 实时检查处方：开始考虑职责间的关系，并发现新职责

继续以同样的思维方式解决问题。如图 8-19 所示，PASS 系统自动检查处方，是由 HIS 系统中医生工作站的调用触发的，处方信息也是通过某种方式（例如参数或 XML 文件）从 HIS 医生工作站获得的。

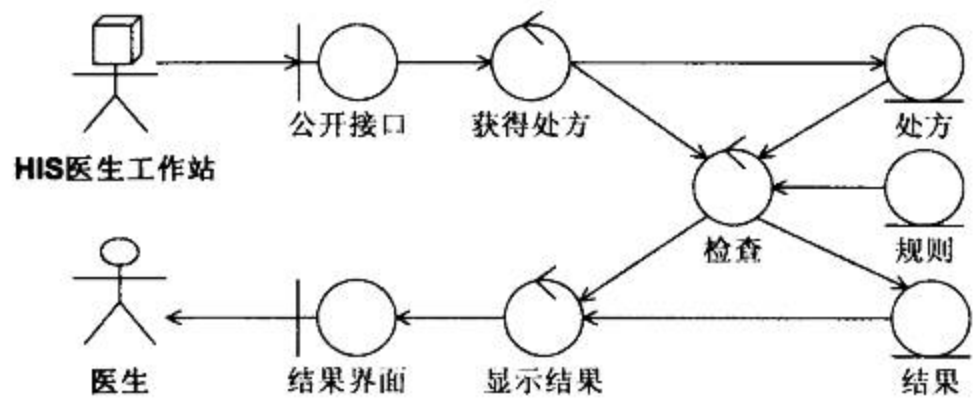


图 8-19 实时检查处方：继续考虑职责间的关系，并发现新职责

实时检查处方最终的鲁棒图如图 8-20 所示，它又进一步考虑了“记录违规用药”这一具体功能场景的支持。

如前文所述，概念架构设计时推荐只对关键功能进行鲁棒图建模。例如，另一个关键功能“自动更新用药规则”的鲁棒图如图 8-21 所示。

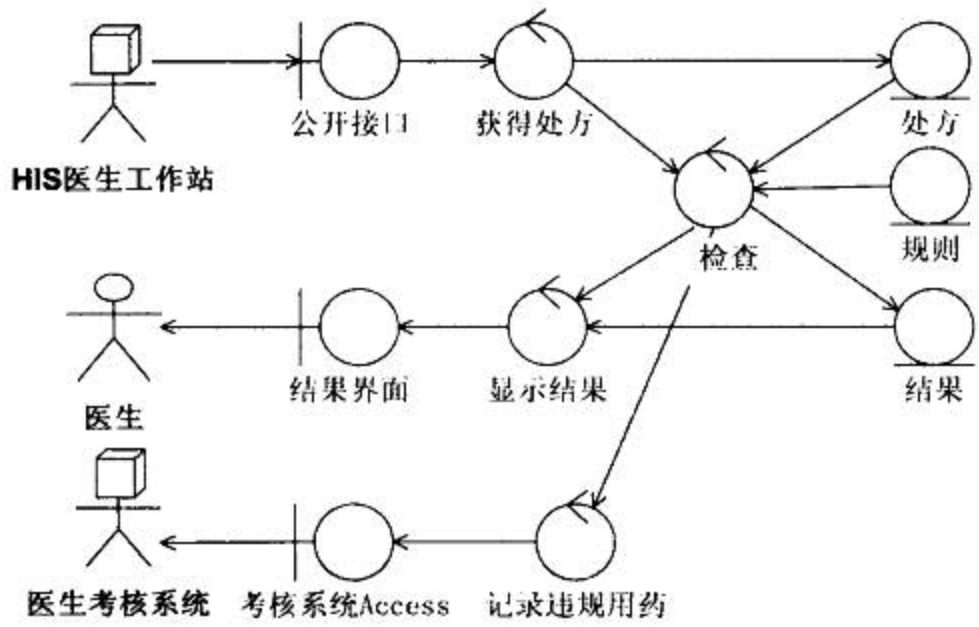


图 8-20　实时检查处方：直到模型比较完善

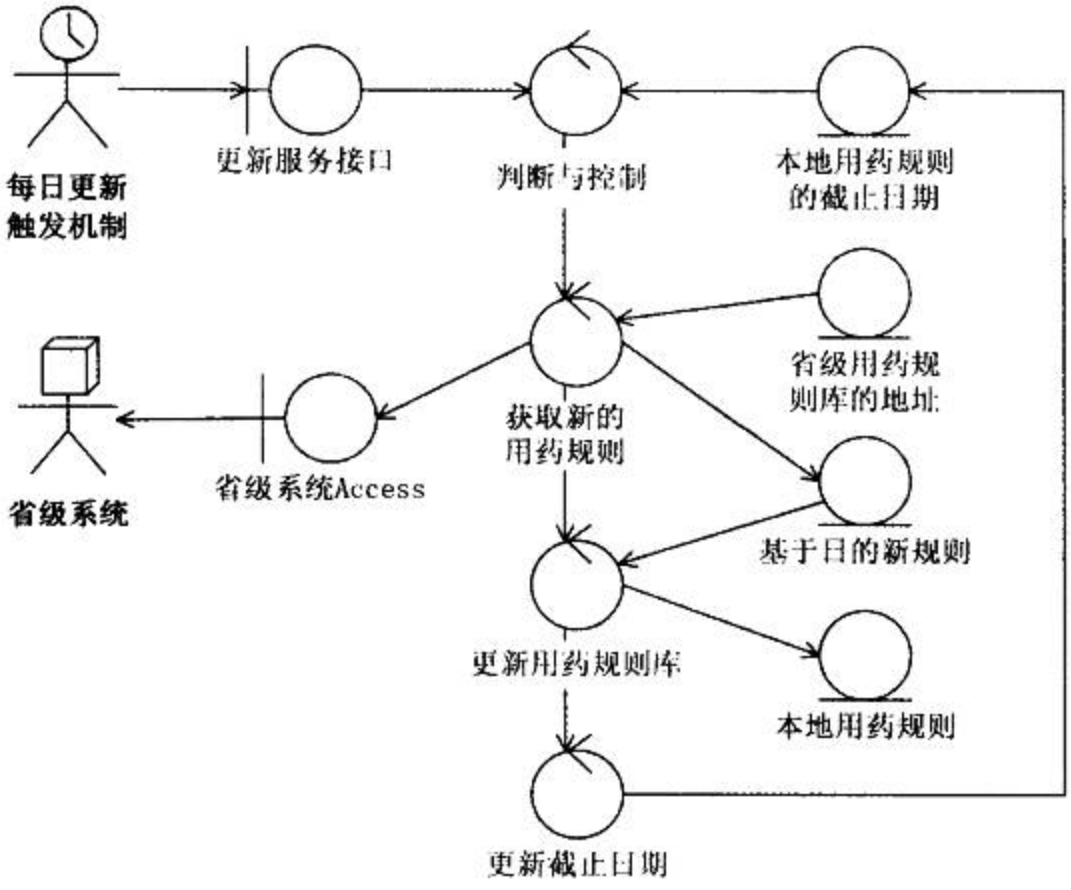


图 8-21　自动更新用药规则：基于鲁棒图的初步设计

第9章
高层分割

复杂性是层次化的。

——Frederick P. Brooks，《人月神话》

分析与综合是思维方向相反的过程。一般是先分析后综合，没有分析就不能有综合；没有综合的分析，也只是片面的分析。

——肖纪美，《梳理人、事、物的纠纷：问题分析方法》

“架构 = 模块 + 接口”的做法，其不足可概括为两点。

第一，忽视了多视图。“模块 + 接口”仅是逻辑架构设计视图的核心内容，而软件系统的架构设计还可能涉及开发视图、运行视图、物理视图、数据视图等多方面的考虑。具体设计方法，请读者参考本书的“第 3 部分 Refined Architecture 阶段”。

第二，忽略了概念架构设计。对规模稍大的系统而言，都必须先根据重大风险（包含功能方面、质量方面、约束方面），有针对性地制定包括“高层分割”在内的设计决策，然后才是“模块 + 接口”一级的设计。

那么，如何对软件系统进行“高层分割”呢？这属于 Conceptual Architecture 阶段第 2 步的工作，正是本章的主题。

9.1 高层分割的两种实践套路

本节就来说明架构设计中“高层分割”的两种实践套路（如图 9-1 所示）：

- 切系统为系统。
- 切系统为子系统。

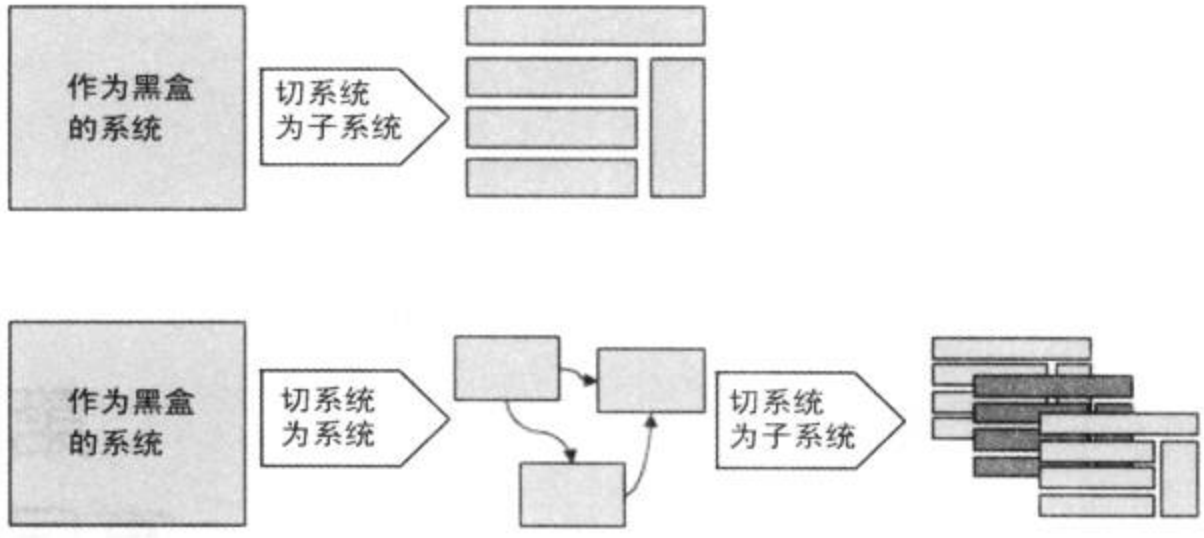


图 9-1　高层分割的两种实践套路：“一步到位”还是“两步到位”

在一线实践中，经常出现这两种方式。虽然从理论上，设计是层层嵌套展开的——即“每一级的子系统”对“下一级的子系统”而言都是系统，都可以单独定义需求和进行设计——但本书比较强调“两种套路”的思维。也就是说，作为一线架构师：

- 你要么告诉自己：我面对的是 1 个“系统的系统（System of Systems）”。
- 要么告诉自己：我要把这个“原子系统”切成若干子系统。

9.1.1　切系统为系统

“切系统为系统”是一种缩略的说法，具体是指：

- 系统比较复杂，须要进行两级高层切分。
- 首先，把系统切成更小一级的系统，每个更小一级的系统都可以有单独的需求、设计、实现……
- 之后，针对每个“更小一级的系统”进行“切系统为子系统”……

这样做的现实意义是巨大的。例如，面临比较复杂的软件系统，很多企业都有“这个项目要设几个架构师”的困惑。图 9-2 给出了本书的建议做法——以电信或广电领域的 BOSS 系统为背景。

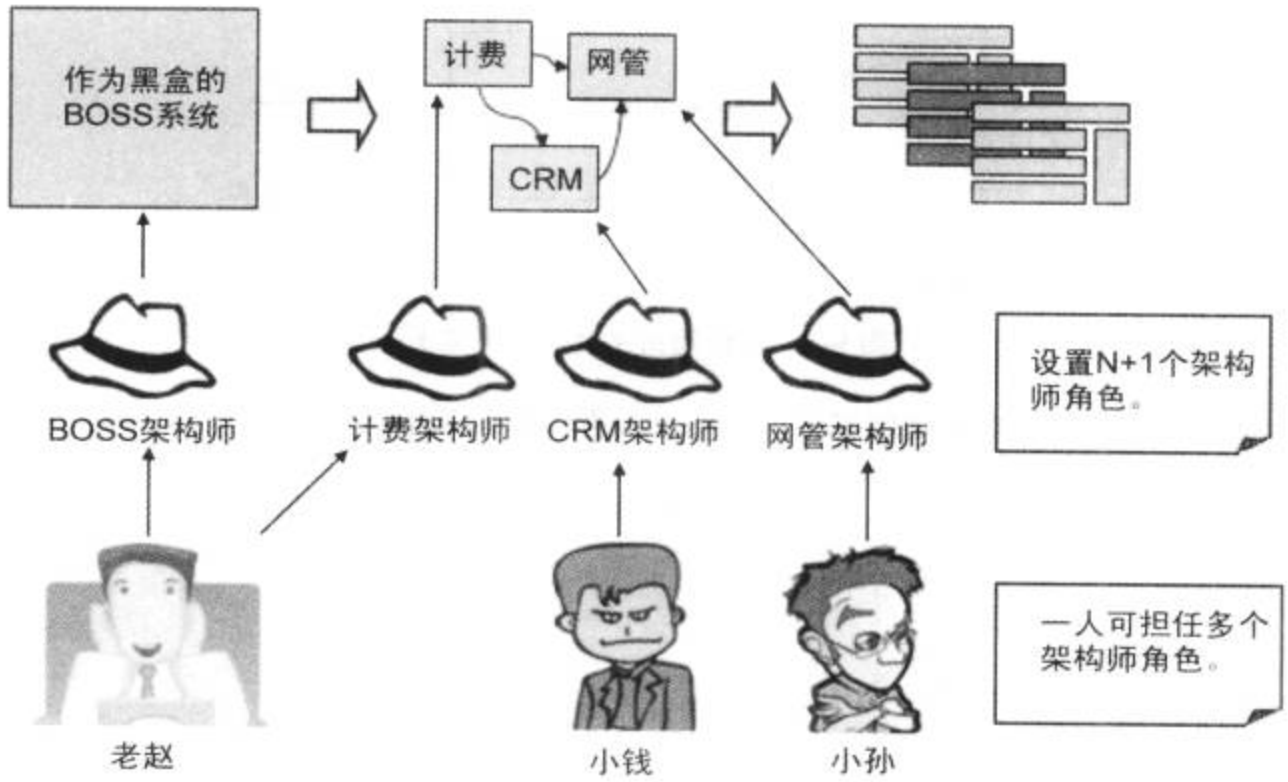


图 9-2 “切系统为系统”时的架构师设置

在实践中，当面临如下两种情况时你须要考虑“切系统为系统”：

- 当系统覆盖的功能范围比较广泛时。

例如，BOSS 系统涉及的需求范围广泛——涵盖网络管理、服务开通、计费、客户关系管理等，于是依“切系统为系统”的做法分为网管系统、服务受理系统、计费系统、客户关系管理系统等。

- 当系统须要部署在比较复杂的硬件环境中时。

很多城市基础服务行业、大型机构、大型企业，都很重视“一卡通”的应用。例如，若一卡通系统要无缝地支持 HR 管理、出入控制、后勤保障等方面的功能，就必须充分考虑复杂的硬件环境给架构带来的冲击——硬件的种类涉及考勤机、通道机、闸机、自助查询机、PC 机、服务器等，于是应当合理规划组成整个一卡通系统的嵌入式应用、桌面应用及 Web 应用等。

9.1.2 案例：SAAS 模式的软件租用平台架构设计

再举一例，来说明如何进行“切系统为系统”式的高层分割。

案例背景

SAAS 风潮又起（以前叫 ASP），“软件租用平台”为用户提供统一的软件租用服务。图 9-3 所示为其上下文图。

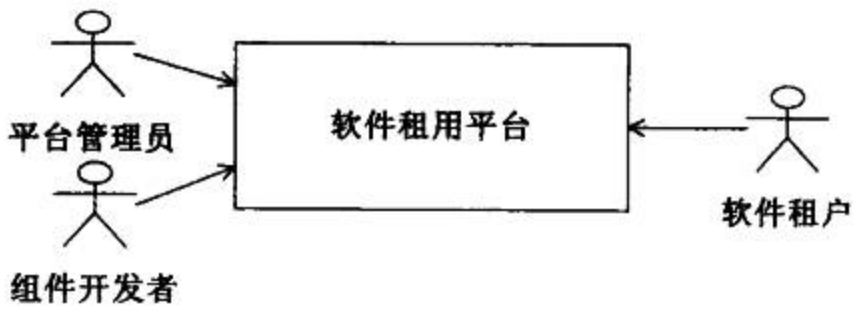


图 9-3　软件租用平台：上下文图

软件租用平台提供的高层功能描述，见图 9-4 所示的业务用例图。

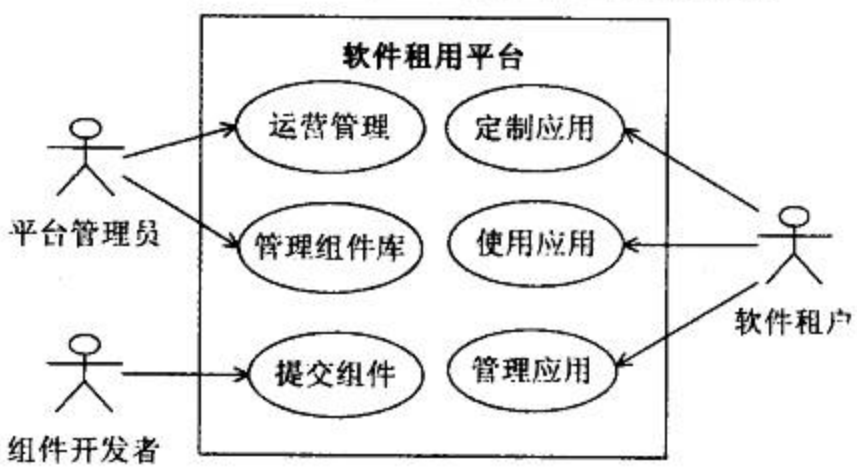


图 9-4　软件租用平台：业务用例图

高层分割的思维过程

架构师心中思量：业界有数据表明，需求的复杂程度每增加 25%，解决方案的复杂程度就增加 100%（如图 9-5 所示）。那么，我如果应该把“软件租用平台”切成三四个相对独立的系统而没切，就意味着人为地制造灾难——太多问题相互杂糅在一起，造成解决方案的复杂程度陡增。嗯，看来须要首先做“切系统为系统”的工作。

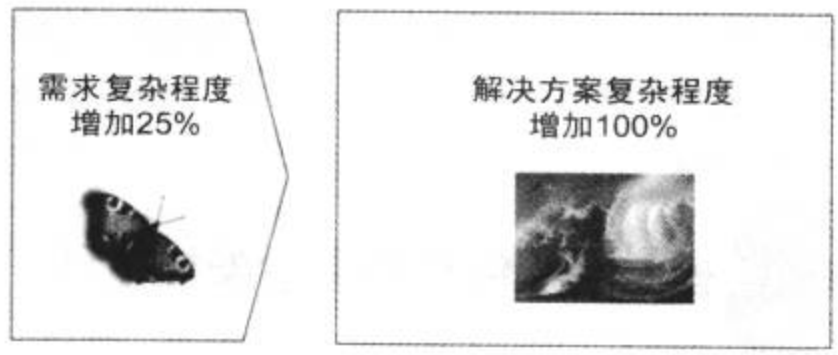


图 9-5　1.25 法则（数据来源：《软件工程的事实与谬误》）

借助鲁棒图，初步识别功能背后的职责，就可以规划高层切分的具体方式。架构师的思维过程如图 9-6 所示。

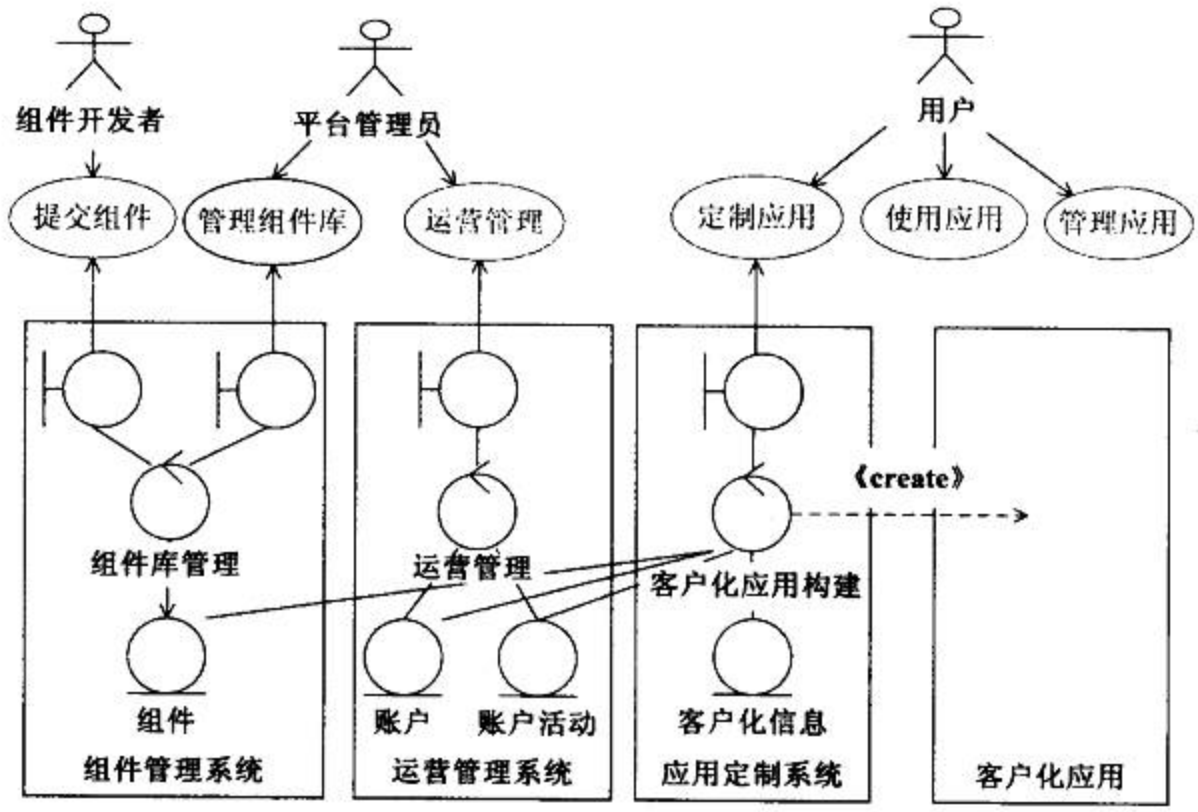


图 9-6 软件租用平台：切系统为系统的思维过程

结果呢？1 个系统被切为 3 个系统——组件管理系统、运营管理系统、应用定制系统。如图 9-7 所示。分别设计和开发这 3 个单独的系统，比直接把软件租用平台当成 1 个 system 开发的可控性高多了。复杂性是根本问题（《人月神话》语），虽无法降低，但可以控制。

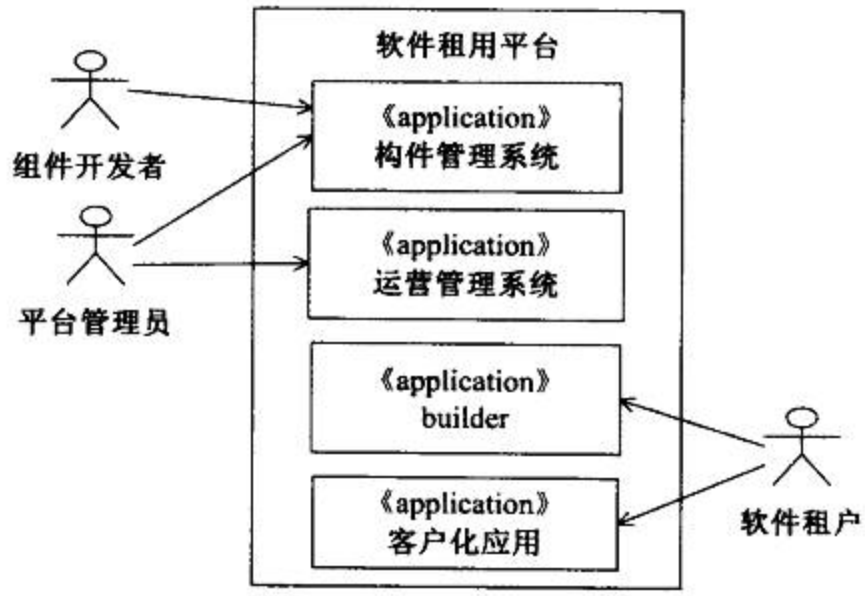


图 9-7 软件租用平台：1 个系统被切为 3 个系统

9.1.3 切系统为子系统

这种做法相当经典，无须太多铺垫。在实践中，最常见的方式就是分层。下面以 PM 系统的层次划分为例进行说明。

案例背景

图 9-8 所示的用例图描述了 PM 系统的功能。

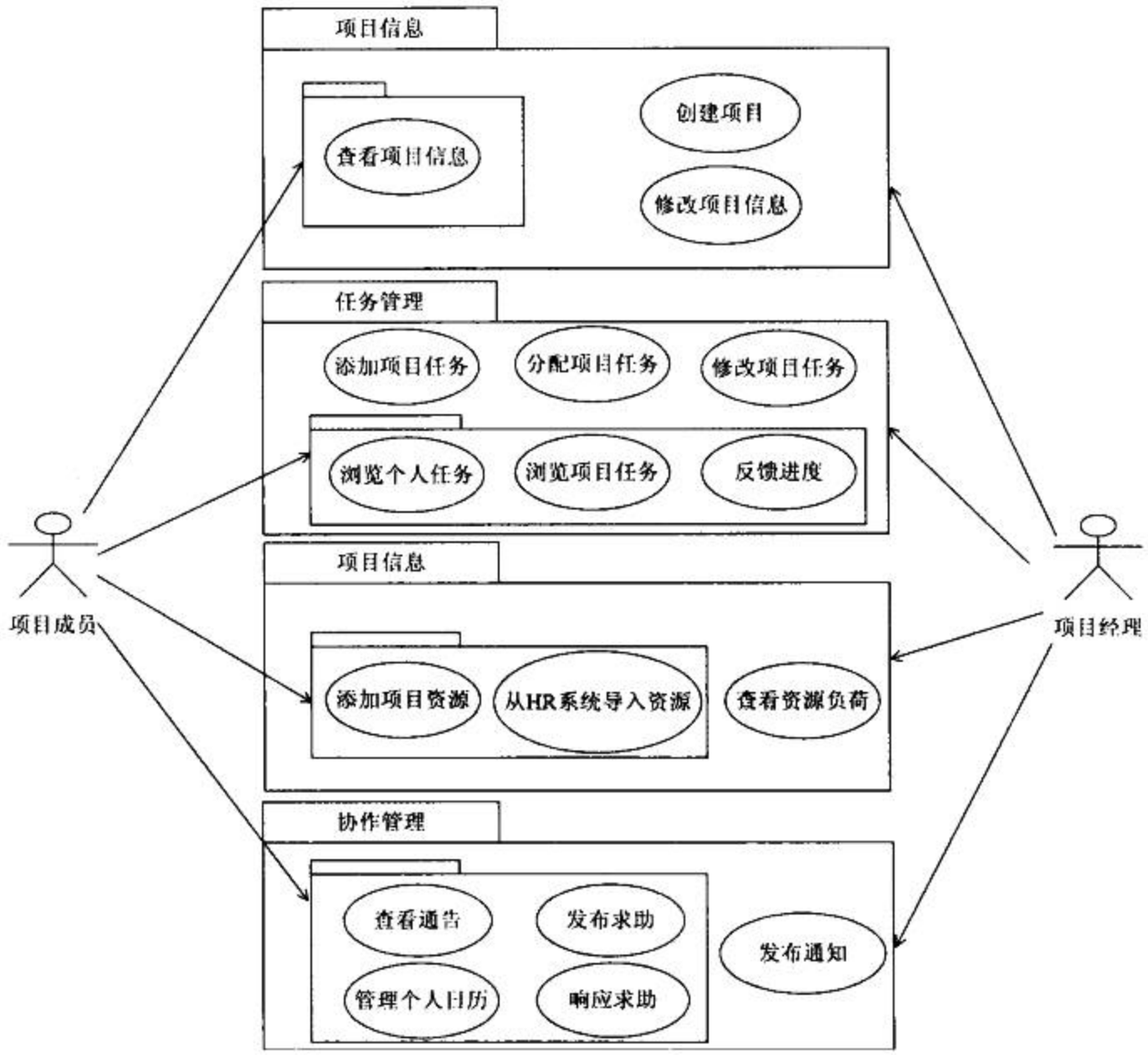


图 9-8 PM 系统：用例图

高层分割

PM 系统的高层分割方案，采用了经典的 4 层架构方式，如图 9-9 所示。

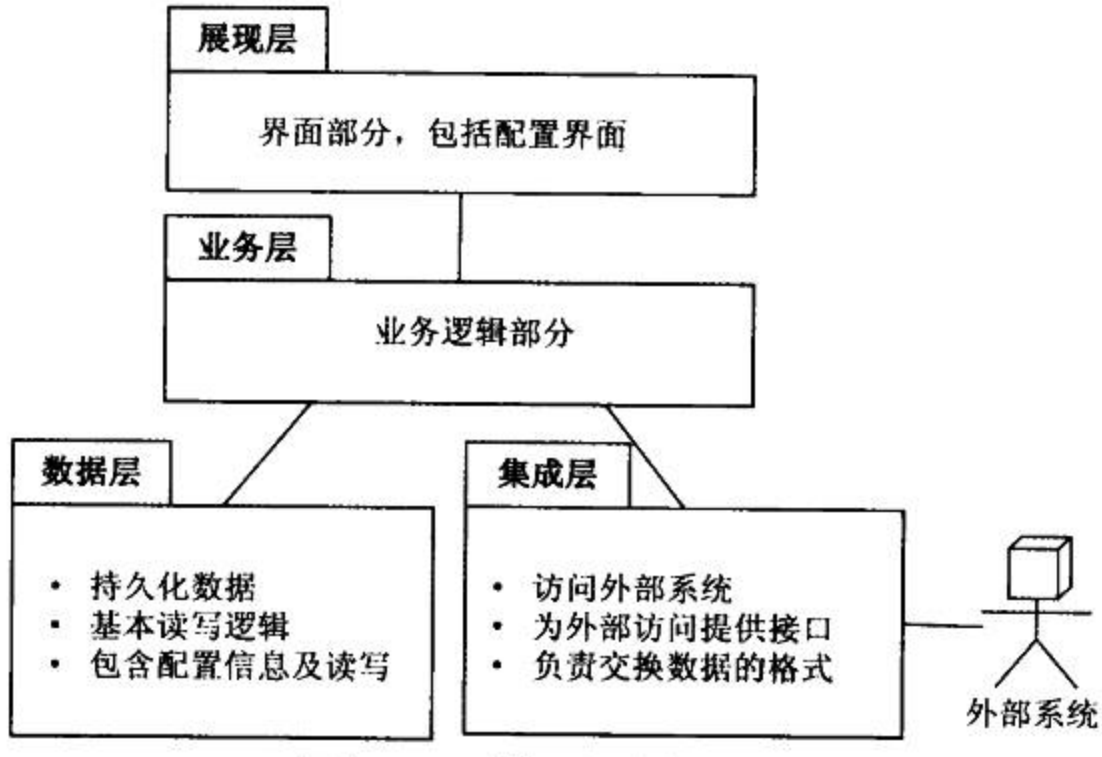


图 9-9 PM 系统：切系统为子系统

9.2 分层式概念架构实践

熟悉的地方没有风景？其实不尽然。架构师们千万别用这句话作为不深入研究分层架构的理由。

人们常说，“分层是最流行的架构模式”。从字面上理解，这似乎意味着大家所进行的“分层”在思想层面上是一致的。但事实并非如此。在实践中，分层有不同的角度，而且互不矛盾。笔者常将之总结为“3+1 种”流派。

- Layer：逻辑层。
- Tier：物理层。
- 按通用性分层。
- 技术堆叠。

9.2.1 Layer：逻辑层

逻辑层（Layer）重视职责的划分，职责之间常常是上层使用下层的关系——但是根本不关心上层和下层是否“能分布”在不同的机器上。图 9-10 所示为 RDBMS 的分层架构，具体分层方式为 Layer。

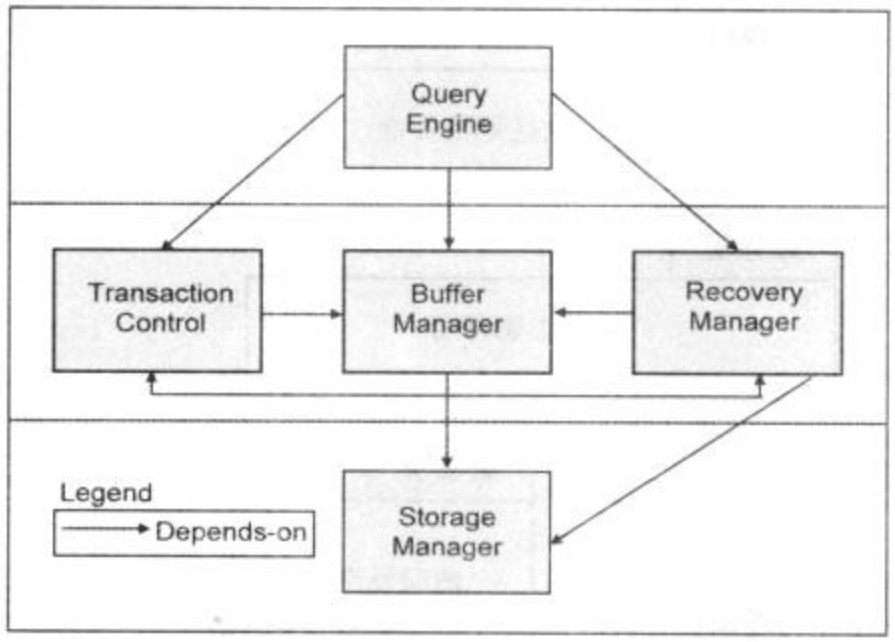


图 9-10 按 Layer 分层：RDBMS

（图片来源：*http://www.swen.uwaterloo.ca/~rekram/files/cs798_assignment_1.htm*）

为对比分析起见，我们问自己如下两个问题：

- 图中查询引擎层（Query Engine），对下层 Buffer 管理（Buffer Manager）部分的访问，是一种跨机器的远程访问吗？

答案是：不知道，不关心。整个架构图中的箭头表示的是逻辑上的服务使用关系，而对物理角度是否是跨机器的访问方式并不关心。稍后我会解释 Tier 分层的概念，读者将会看到 RDBMS 案例所采用的分层架构不是后面要讲的按 Tier 分层。

对比结果：按 Layer 分层 ≠ 按 Tier 分层。

- 图中的查询引擎（Query Engine）、Buffer 管理（Buffer Manager ）和存储管理（Storage Manager ）是通用性逐渐增加吗？（ “通用程度越大，所处层次就越靠下”是按通用性分层的常见方式。）

答案是无法确定哪一层更通用。例如，作为最下层的存储管理层本来支持硬盘，但后来要支持磁盘阵列，再后来要支持 SAN（存储区域网络），这都要求存储管理层有针对性地进行改变——看来，存储管理层并不通用。

对比结果：按 Layer 分层 ≠ 按通用性分层。

9.2.2 Tier：物理层

物理层（Tier）指“能分布”在不同机器上的软件单元，不同物理层之间必须有跨机器访问的能力——可以通过远程调用、或通信协议等方式。图 9-11 所示为 Java EE 的分层架构，具体分层方式为 Tier。

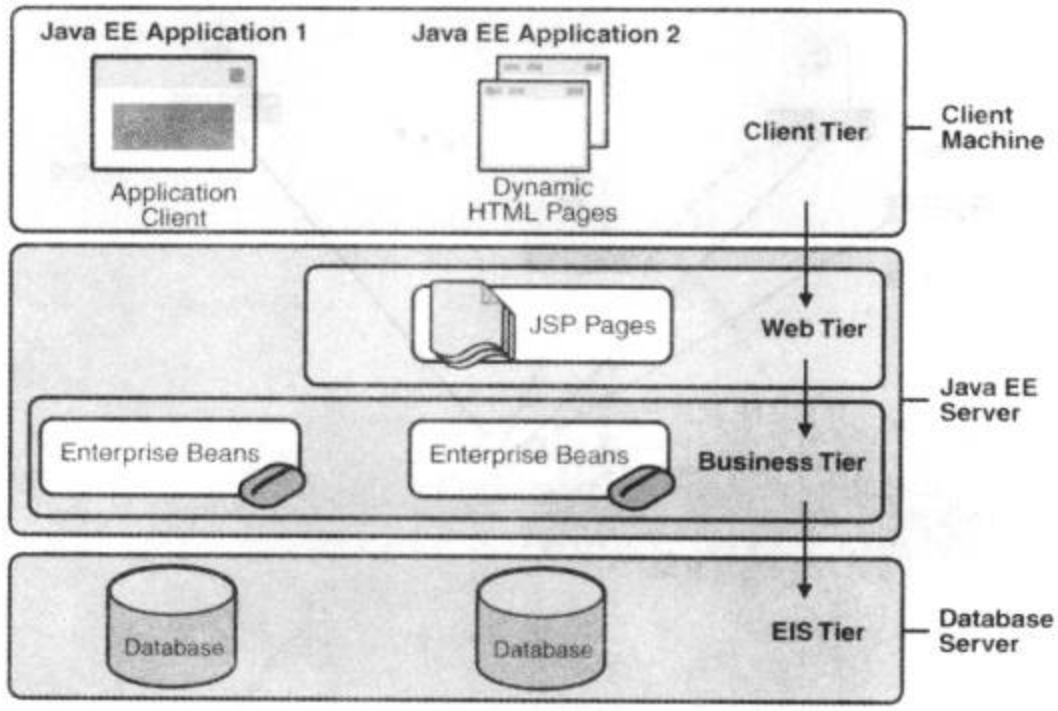


图 9-11 按 Tier 分层：Java EE（图片来源：《The Java EE 5 Tutorial》）

关于 Tier 这种分层方式，最须要强调的是，几层（Tier）架构是看“能分布”的能力，不是看“实际部署情况”。下面来分析一段来自网上的话：

我们通常说的 Java EE 应该是 N-layer 的，因为从逻辑上来看，Java EE 里面有表现层、业务逻辑层和数据永久层。从物理上而言，这 3 层可以在不同的 tier 上（表现层在 PC 上，业务逻辑层在应用服务器上，数据永久层则在数据库服务器上），也可以在一个 tier 上，比如 Martin 说的，如果把数据库、应用服务器和浏览器都装在一台笔记本电脑上，那么，3-layer 就在 1-tier 上了。

这段话恐怕问题不小！

毕竟，“N-Tiers 架构”的一大好处是可伸缩性——业务量小的时候将 N 个 Tier 都部署在同一台机器上，等业务量大的时候再为每个 Tier 单独安排一台或一组机器，这恰恰是“N-Tiers 架构”的目标！所以，一个系统如果架构设计时是“4-Tiers 架构”的，并且开发时也实现了这一点，那么把它们部署到同一台机器上并没有改变“4-Tiers 架构”。最终，工程师的实际部署方案决定系统是几层（Tier）架构，这未免荒唐。

其实，总结出“3 级”映射关系（而不是“两级”）就清楚了：

逻辑层 Layer ——> 物理层 Tier ——> 一台或一组计算机

关于按 Tier 分层，再看一例：中国电信全球眼系统（图 9-12）。很多人不理解，这也是多 Tier 架构呀？答案是肯定的。监控点层、平台层、客户端层之间必然是以能进行跨机器访问的协议互相通信的，所以照样是按 Tier 分层的架构——只不过每个 Tier 的部署规模比较大罢了。

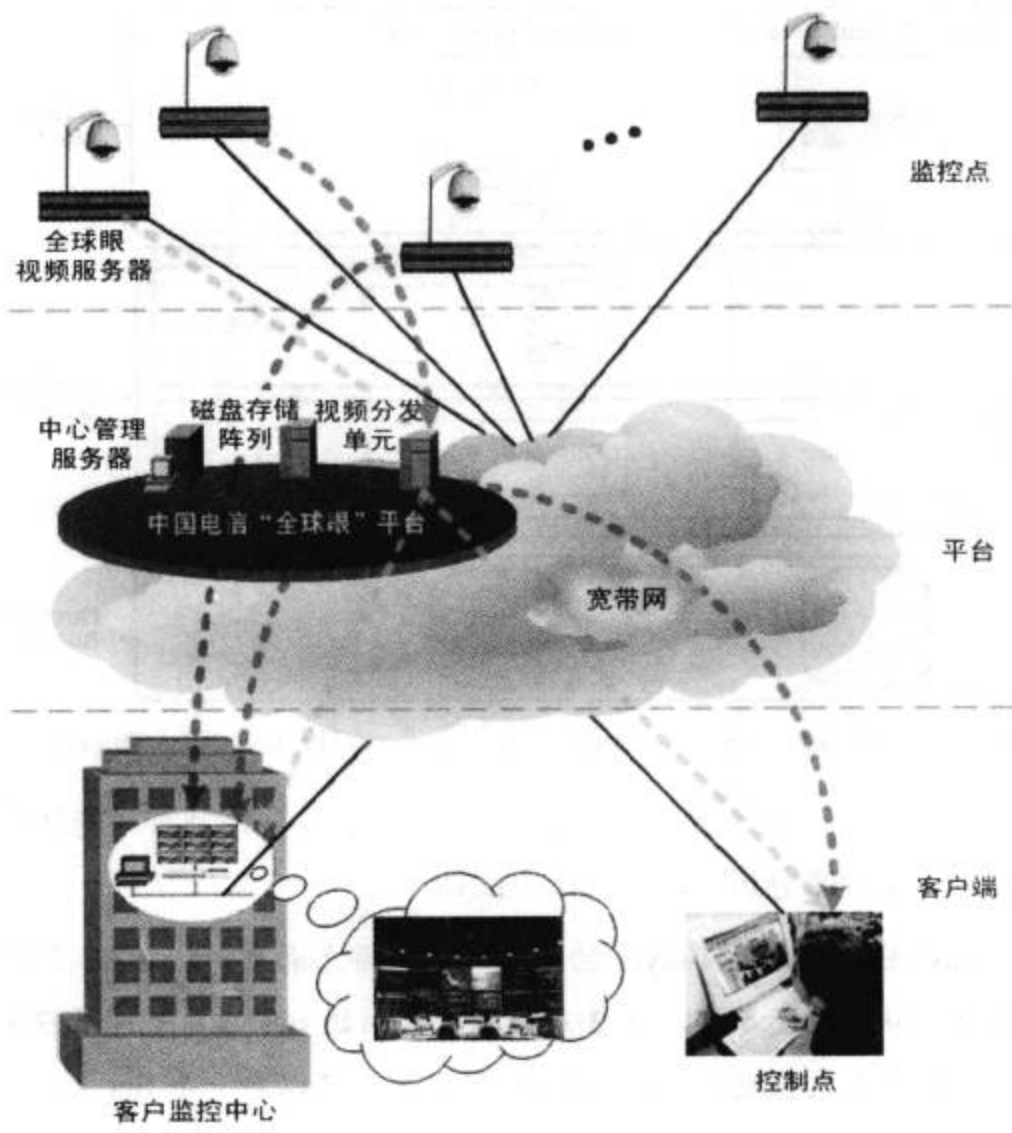


图 9-12　按 Tier 分层：全球眼（图片来源：中国电信“全球眼”资料）

9.2.3　按通用性分层

严格来讲，按通用性分层是另一种 Layer，但是，绝对有必要让它“独立门户”以引起实践者的足够重视。

按通用性分层是指：将通用性不同的部分划归不同的层，以此作为系统的总体切分方式。

一般而言，通用程度越大，所处的层次就越靠下。例如图 9-13 所示的架构，按通用性不同分为 4 层：应用特定层、业务相关层、中间件层、系统软件层。

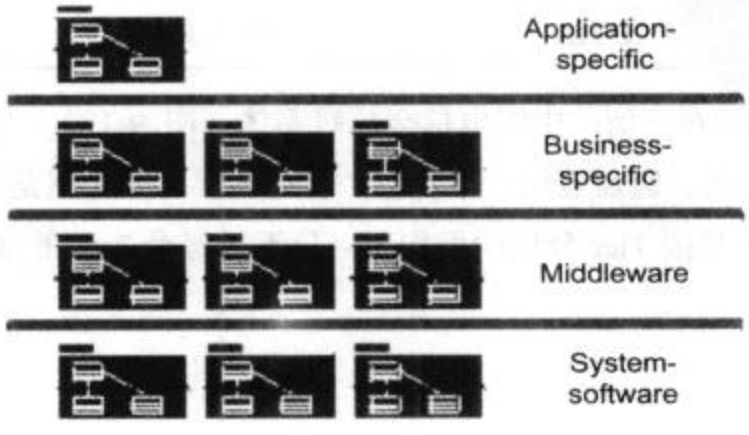


图 9-13　按通用性分层（图片来源：RUP）

不过，嵌入式系统的分层架构方式有所不同（如图 9-14 所示）：通用性最强的层（图中的操作系统层、中间层）位于中间，硬件相关部分，以及应用特定部分分别位于下层和上层。其实，这种“中间通用、上下专用”的分层方式对可移植性关键的通信系统、控制系统、软件平台等情况都非常重要。

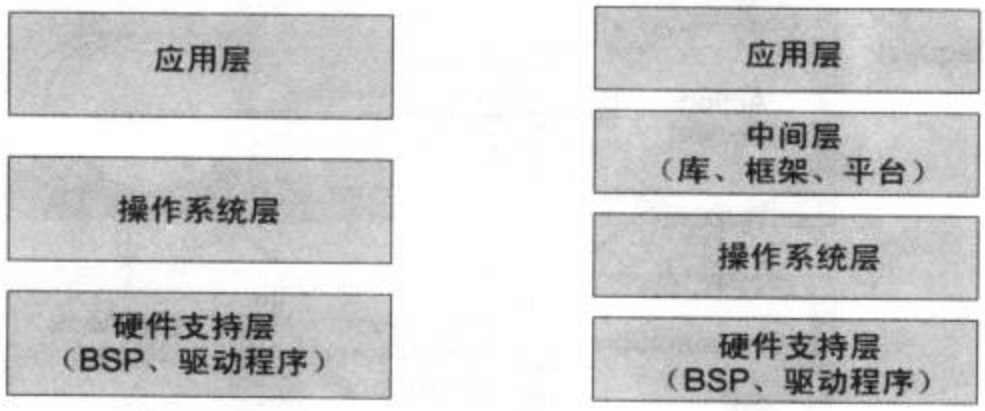


图 9-14 按通用性分层（嵌入式系统情况）

9.2.4 技术堆叠

技术堆叠不是独立的架构模式，而是基于分层架构（或其他架构模式）提供的进一步说明。图 9-15 是一个例子。其中，基本的架构模式是分层——按 Tier 分层，并明确了各个技术点。

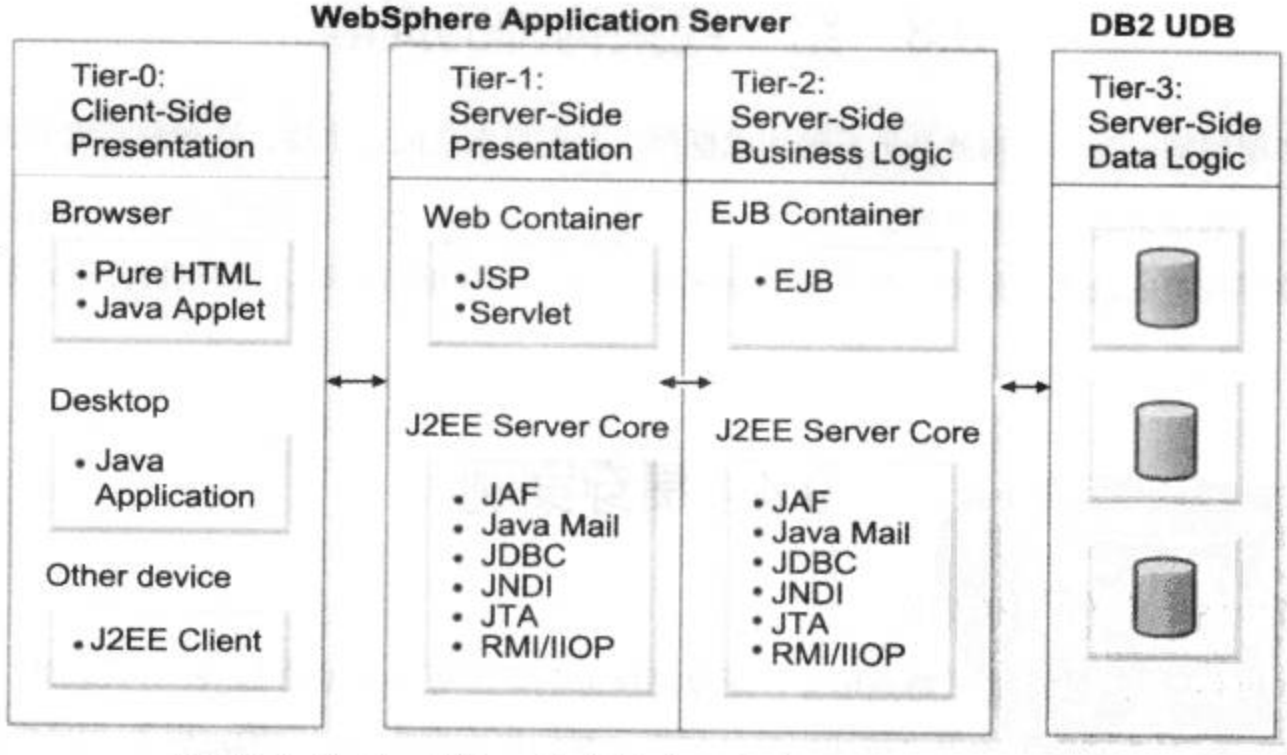


图 9-15 按 Tier 分层 + 技术堆叠（图片来源：J2EE 相关资料）

图 9-16 是另一个例子。其中，基本架构模式是 MVC，也加入了技术堆叠的描述。

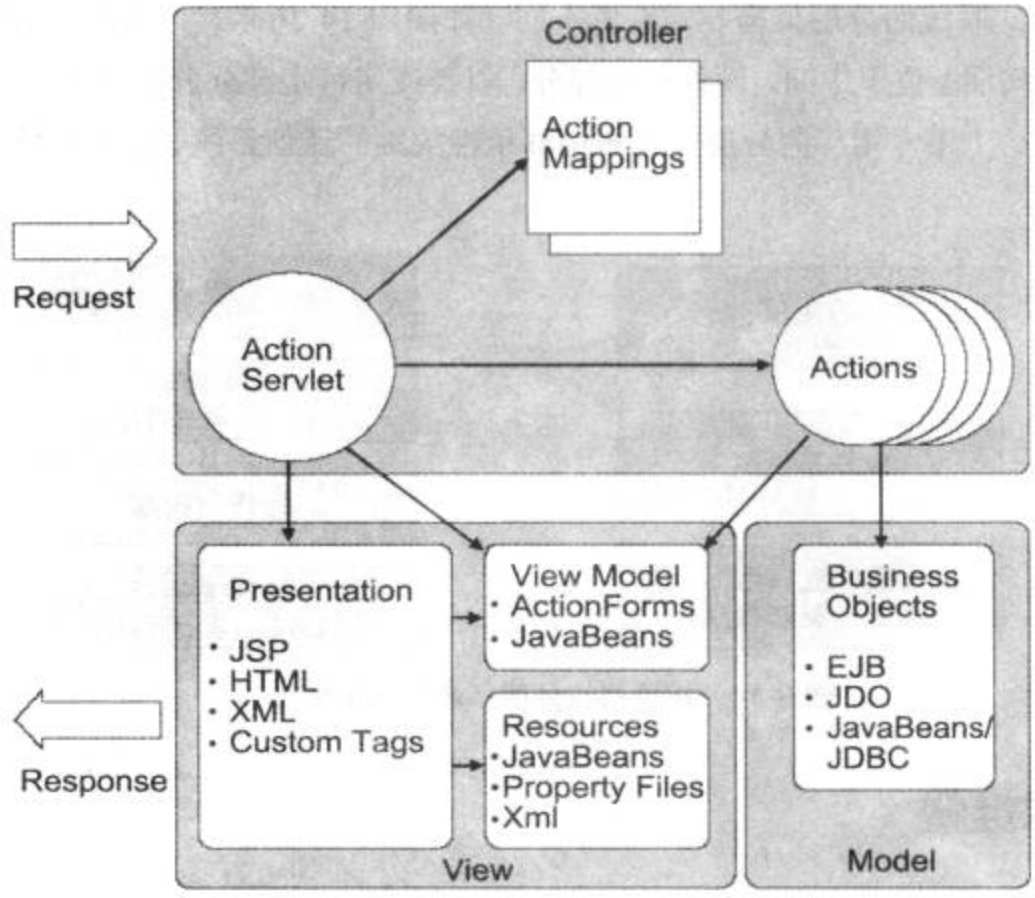


图 9-16　MVC + 技术堆叠（图片来源：Struts 相关资料）

9.3　给一线架构师的提醒

许多架构师认为，架构就是把系统切成框框，再在框框之间连上线。这种观点太片面。

高层分割很重要，但不是概念架构的全部。除了切分决策之外，概念架构还包括技术选择、权衡策略等种类的决策。例如，为了支持各种相互矛盾的非功能需求，仅调整切分方式是远远不够的。

9.4　贯穿案例

继续 PASS 系统的贯穿案例。

由于 PASS 系统的分布式特点明显，所以高层分割除了考虑常见的 Layer 方式之外，我们也应考虑 Tier。而通过主动考虑“按通用性分层”，我们也大有收获——确定引入“PASS 系统医生模块通用 SDK”而使无谓的重复开发工作量降至最低。

9.4.1　从初步设计到高层分割的过渡

回顾第 8 章的初步设计成果，通过对这些已发现的职责进行“综合”，可以确定系统基本的

高层分割方式。如图 9-17 所示。

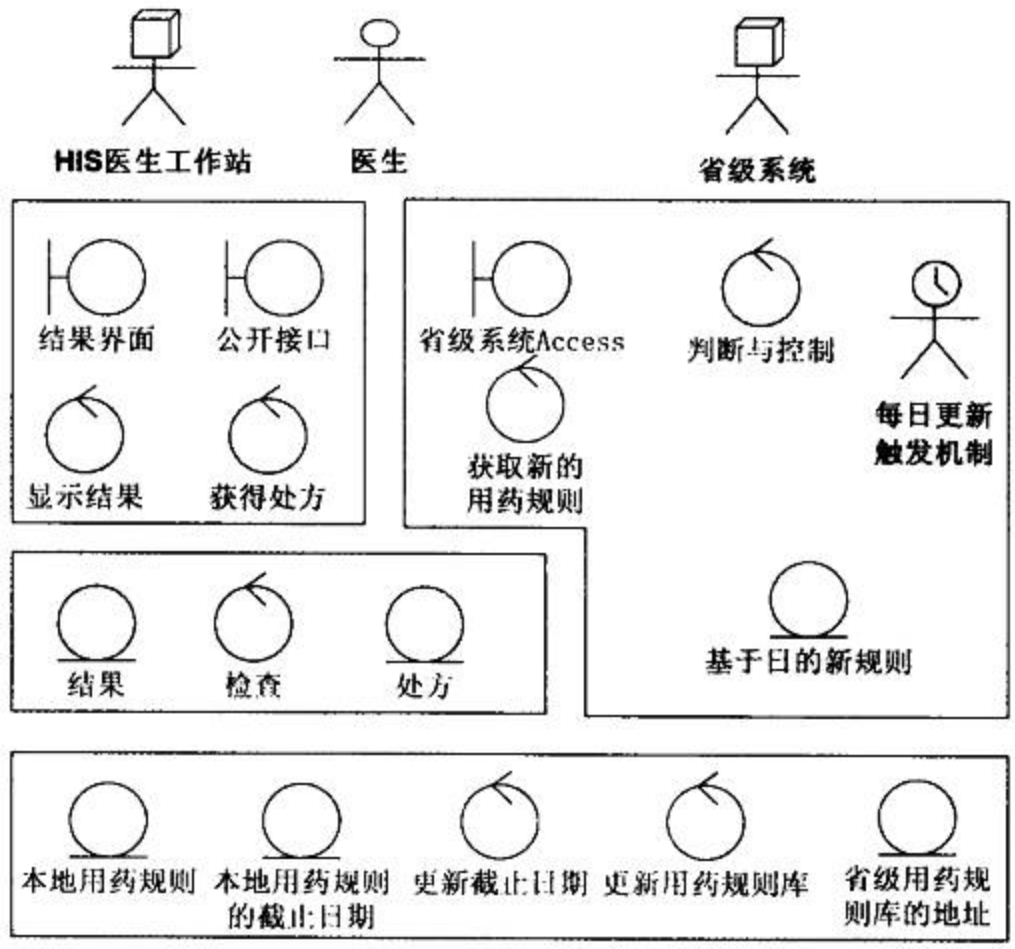


图 9-17 从初步设计到高层分割的过渡

9.4.2 PASS 系统之 Layer 设计

于是，可以得到按 Layer 对 PASS 系统进行高层分割的方式。如图 9-18 所示。

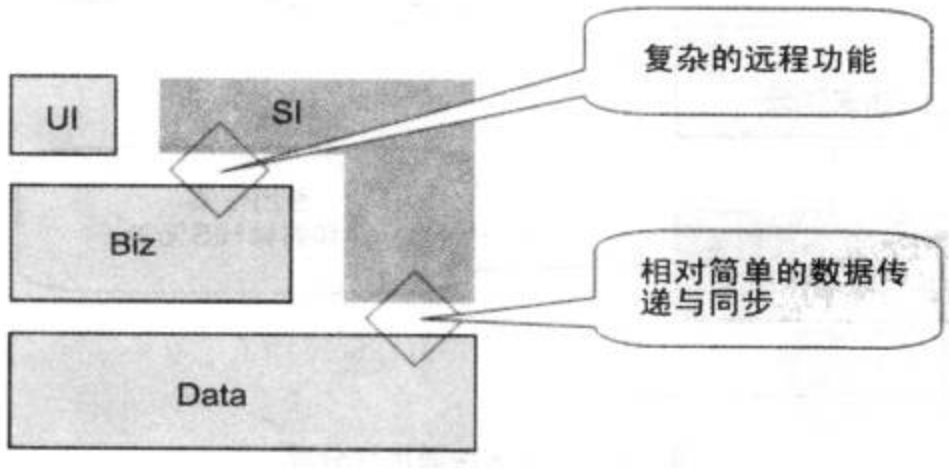


图 9-18 PASS 系统之 Layer 设计

9.4.3 PASS 系统之 Tier 设计

按 Layer 切分未反映 PASS 系统很强的分布式特点，我们应进一步从 Tier 角度考虑 PASS 系统的高层分割方式。如图 9-19 所示。

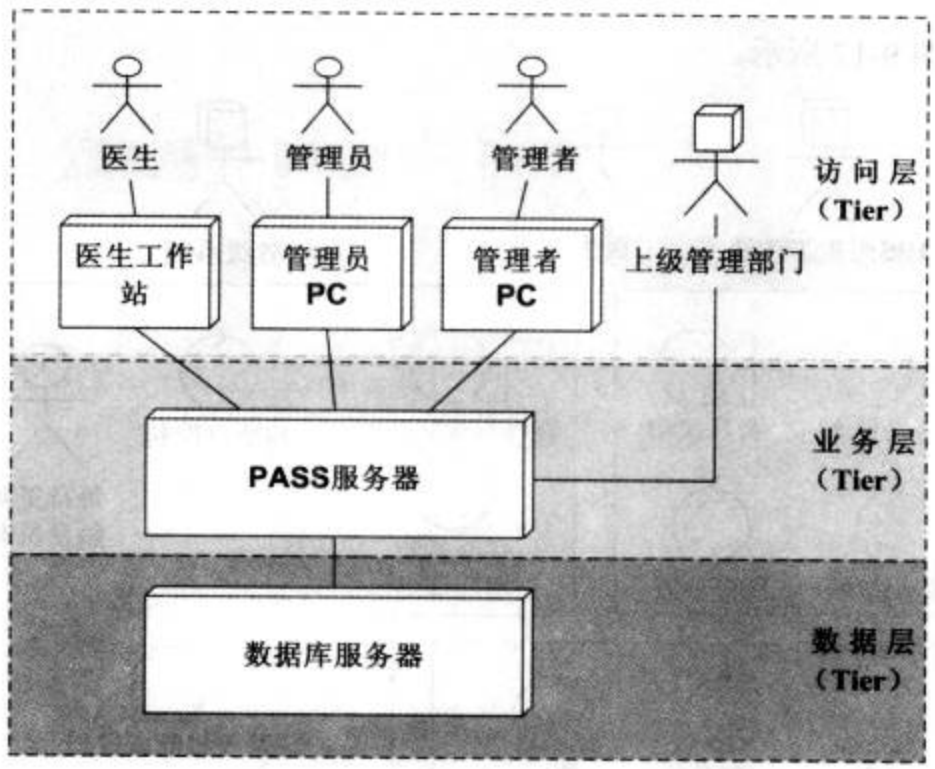


图 9-19　PASS 系统之 Tier 设计

9.4.4　引入通用性分层

如果进一步质疑“可重用性”（详见第 10 章“贯穿案例”部分的目标—场景—决策表），将想到应将嵌入 HIS 的程序进一步切分——分出“PASS 系统医生模块通用 SDK”，它和任何具体 HIS 无关，所以通用性高，应分离出去。如图 9-20 所示。

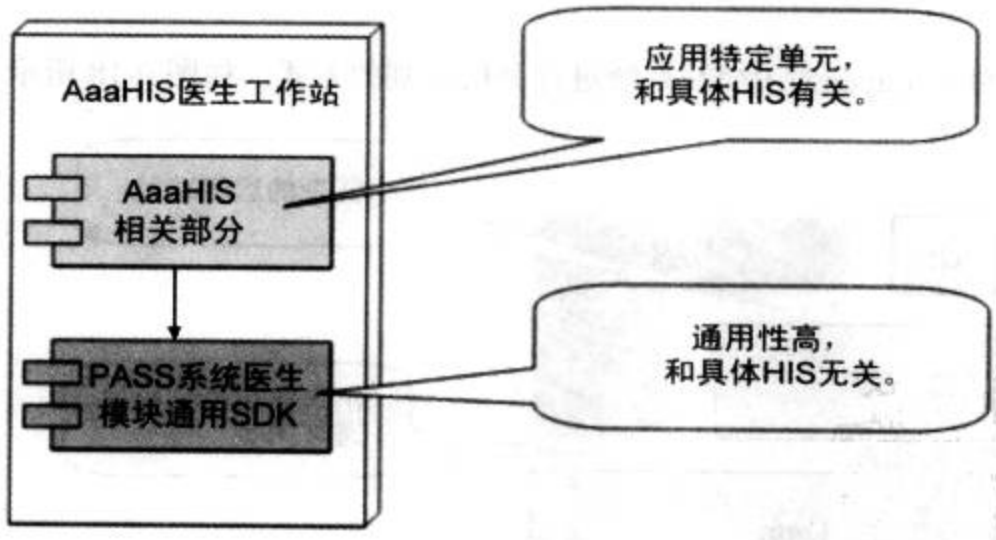


图 9-20　引入按通用性分层

第 10 章
考虑非功能需求

架构不仅仅是系统功能需求的结果。

——Len Bass，《软件构架实践（第 2 版）》

在我们当中，有不少人一厢情愿地认为：只要所开发出的系统完成了用户期待的功能，项目就算成功了。但这并不符合实际。

——温昱，《软件架构设计》

《软件架构设计》一书曾指出，“其实任何作为复合整体的复杂事物都可能有架构，比如一本书”。“非功能目标的考虑”在 ADMEMS 方法中不是一个阶段，而是一个纵穿环节。为了避免内容重复，本书将在“第 4 部分 专题：非功能目标的方法论”集中讨论如何针对非功能需求进行理性设计。

本章重点是贯穿案例——PASS 系统概念架构设计的第 3 步：考虑非功能需求。

10.1 考虑非功能目标要趁早

概念架构 ≠ 理想化架构。ADMEMS 方法强调：

- 重大需求塑造概念架构。这里的“重大需求”应涵盖功能需求、质量及约束 3 类需求中的关键部分。
- 概念架构是一个“架构设计阶段”，必须在细化架构设计阶段之前，针对重大需求、特色需求、高风险需求，形成稳定的高层架构设计成果。
- 如果只考虑“功能需求”来设计概念架构，将导致概念架构沦为“理想化架构”，这个脆弱的架构不久就会面临“大改”的压力，甚至直接导致投标等工作失败。

10.2 贯穿案例

非功能需求往往非常笼统，而场景是一种明确性很强的技术。目标-场景-决策表可以让架构师理性地应对非功能需求。

如表 10-1 所示，通过场景，我们质疑了可重用性的做法。为了避免开发多个完成孤立的“医生工作站嵌入单元”，引入的设计决策是“分离出不变部分”，将“PASS 系统医生模块通用 SDK”提炼出来（如第 9 章图 9-19 所示）。

表 10-1 可重用性的“目标-场景-决策表”分析

目标	场景	决策
可重用性	• 欲嵌入的 HIS 系统种类较多，如何避免开发多个完全孤立的“医生工作站嵌入单元” • ……	研究可能嵌入的 HIS，确定“医生工作站嵌入单元”的不变部分和可变部分 ……

不要忘了架构设计是质疑驱动的——概念架构也经常是经过多次循环的设计结果（回忆图 1-7 ADMEMS 方法推荐的概念架构设计做法）。现在，我们来质疑 PASS 系统的持续可用性，它毕竟是用于辅助医院运营的系统呀！如表 10-2 所示。

表 10-2 持续可用性等的“目标-场景-决策表”分析

目标	场景	决策
持续可用	• 硬盘可能发生故障，避免单点故障造成 PASS 停机	磁盘阵列
	• DBMS 可能出问题，避免单点故障	故障转移群集
	• PASS 系统安全性差，将直接导致持续可用性下降	提高安全性
安全性	• 系统中业务数据，整体毁坏或丢失，对医院影响大	磁带备份
	• ……	……

根据上述基于目标-场景-决策表的思考，我们调整系统的原有架构图，如图 10-1 所示。一线的架构实践一再告诉我们，考虑非功能需求会引起“架构中间设计成果”的调整！

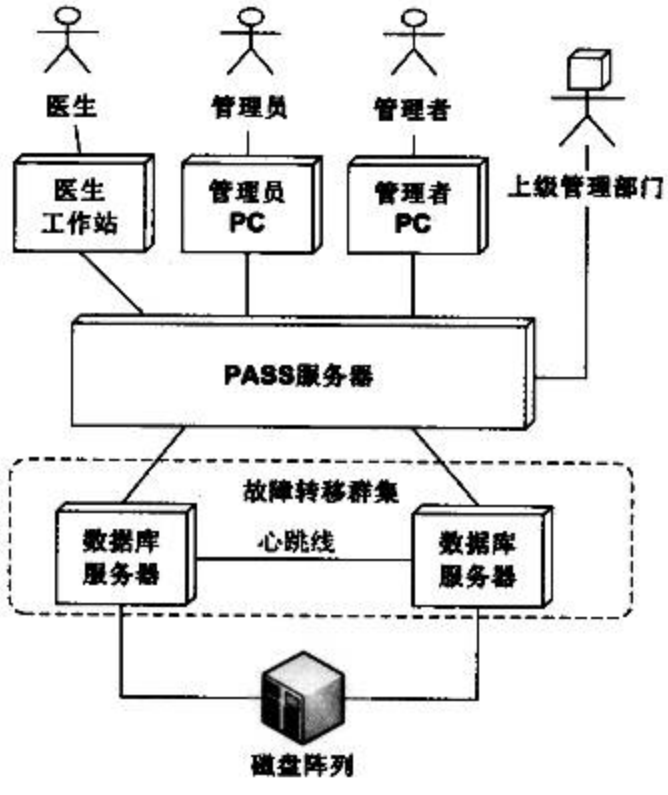
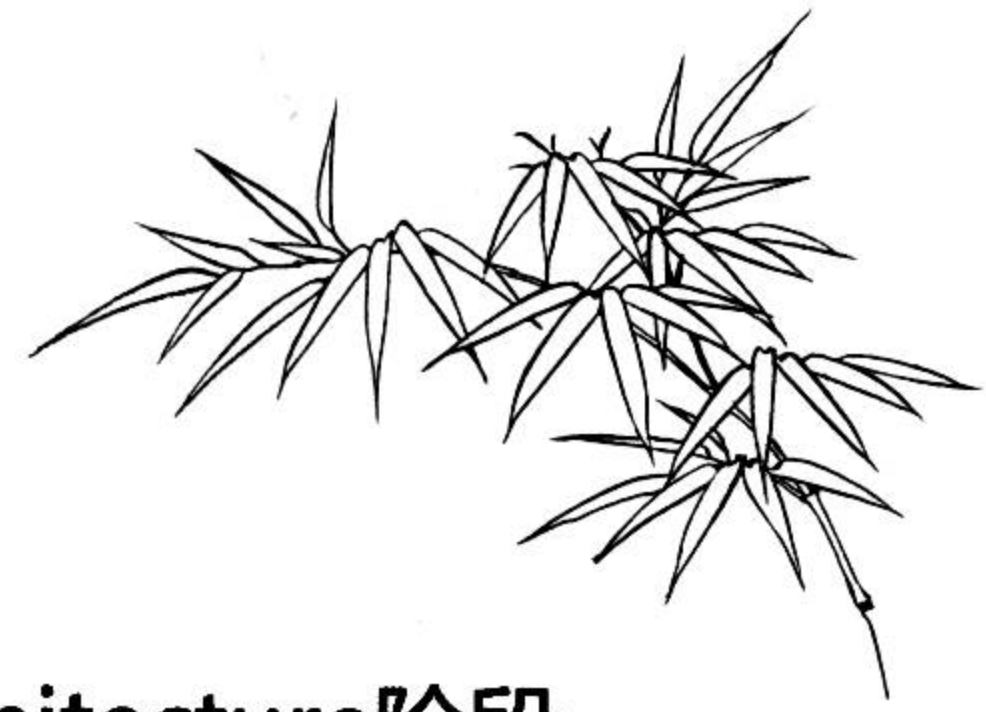


图 10-1 考虑非功能需求会引起“架构中间设计成果”的调整



Refined Architecture阶段

第 11 章
细化架构的故事

如果一个项目的系统架构（包括理论基础）尚未确定，就不应该进行此系统的全面开发。

——Barry Boehm，《Engineering Context》

如果选择视图的工作没有做好，或者以牺牲其他视图为代价，只注重一个视图，就会冒掩盖问题以及延误解决问题（这里的问题是指那些最终会导致失败的问题）的风险。

——Grady Booch，《UML 用户指南》

从概念架构到细化架构，先设计概念架构，构思关键问题的解决策略；再进行细化架构的设计，以保证为开发提供足够的指导和限制……这符合人类解决问题的规律，因此被广泛采用。

但在实际中，细化架构设计还存在很多差强人意之处，甚至经常被忽视。下面两个故事，就发生在我们身边。

11.1 骄傲的架构师，郁闷的程序员

11.1.1 故事：《方案书》确认之后

公司在谈一个项目，但还没有得到客户的认可。后来，除了老李这个项目经理之外，小张作为架构师、小王作为需求分析师也都参与了进来。他们几个在和客户沟通的基础上，通力合作，最后成功提交了《方案书》，并获得客户的认可。《方案书》是老李、小张、小王各负责一部分来写的，其中架构师小张负责总体设计部分。

小张认为：《方案书》被认可说明架构已经很明确，无须“再”架构设计了。

最后，苦了程序员，因为他们在实际开发中没有得到足够的指导和限制。

11.1.2　探究：“方案”与“架构”的关系

究其原因，这是因为概念架构难以支持并行开发。要支持开发组相对独立地进行工作，须要提供指导和限制作用更明确的“规约”一级的设计。

具体而言，细化架构和概念架构之间存在如下典型差异：

- 接口。在细化架构中，接口占据非常核心的地位，而概念架构并不关心明确的接口定义（只有抽象的组件和抽象的交互机制）。
- 子系统。细化架构重视通过子系统和模块来分割整个系统，并且子系统往往有明确的接口；而概念架构中只有抽象的组件，这些组件没有接口，只有职责，一般是处理组件、数据组件或连接组件中的一种。当然，概念架构中也有“大组件分解成小组件”的设计决策，但并非子系统的含义。
- 交互机制。细化架构中的交互机制应是“实在”的，如基于接口编程、消息机制或远程方法调用等；而概念架构中的交互机制是“概念化”的，例如“A层使用B层的服务”就是典型的例子，这里的“使用”到了细化架构中可能基于接口编程、消息机制或远程方法调用等其中的一种。

当然，概念架构和细化架构都满足软件架构的定义——无论是“架构＝组件＋交互”，还是“架构＝重要决策集”。

方案的制定，为什么需要项目负责人、需求人员、架构师等共同参与呢（如图11-1所示）？因为方案涉及的工作内容不仅仅是架构，还涉及项目管理和需求工作。“方案”和“架构”的联系与区别如下：

- 方案包含一定的架构内容。
- 方案涉及的架构基本在概念架构一级。
- 架构设计的工作还远未完成。

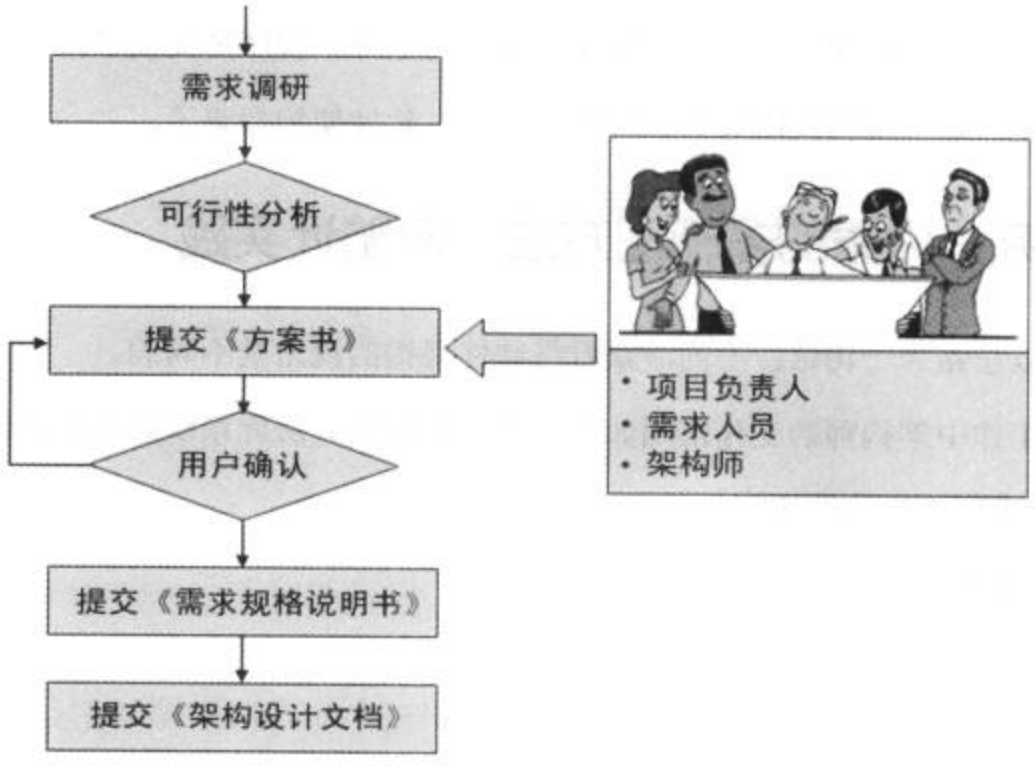


图 11-1 方案制定需要项目负责人、需求人员、架构师等角色的共同参与

所以，架构师应记住：

方案 = “项目 + 需求 + 架构”的总览

方案 ≠ 架构的全部

11.2 办公室里的争论

11.2.1 故事：办公室里，争论正酣

办公室里，关于什么是软件架构，争论正酣。

程序员说，软件架构就是要决定须要编写哪些类，使用哪些现成框架（Framework）。程序经理笑了；

程序经理说，软件架构就是模块的划分和接口的定义。系统分析员笑了；

系统分析员说，软件架构就是为业务领域对象的关系建模。配置管理员笑了；

配置管理员说，软件架构就是开发出来的及编译后的软件到底是个啥结构。数据库工程师笑了；

数据库工程师说，软件架构规定了持久化数据的结构，其他一切都不过是对数据的操作而已。部署工程师笑了；

部署工程师说，软件架构规定了软件部署到硬件的策略。用户笑了；

用户说，软件架构就是决定一个个功能子系统如何划分。程序员又笑了；

大家想了想说，这些架构视图好像我们都需要啊，软件架构师哭了。

11.2.2　探究：优秀的多视图方法，应贴近实践

上述争论可以总结为一句话：不同涉众看待软件架构的视角是不同的。

但是，实际工作中架构师的工作范围如此广泛，多视图方法能系统地涵盖吗？例如：

- 进程、线程的相关设计。
- 接口的定义。
- 子系统的划分。
- 服务器的选型。
- （若你用 C）结构化方法的模块设计“放”在哪里？
- 考虑 Layer（逻辑层）。
- 考虑 Tier（物理层）。
- （基于并行开发的需要）源程序目录结构的定义。
- 数据分布与数据库 Schema。
- （若没选 RDBMS 而选了文件方式）文件格式的定义。
- （嵌入式系统中常将数据保存到 Flash）Flash 存储结构的定义。
- ……

答案是：贴近实践的多视图方法，应将一线架构师的各项具体工作涵盖其中。如图 11-2 所示。

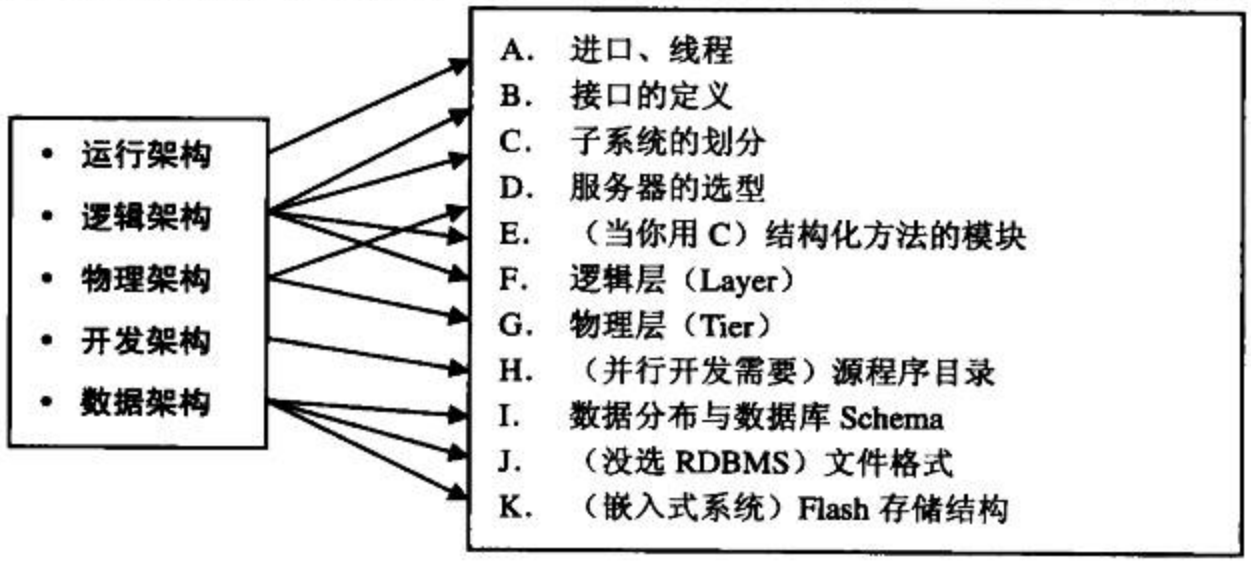


图 11-2　贴近实践的多视图方法，应将一线架构师的各项具体工作涵盖其中

11.3 展望“Refined Architecture 阶段篇”

总结一下。首先，架构设计仅进行到概念架构层面，对支持团队的并行开发而言是远远不够的；常见的错误就是把《方案书》中的概念架构设计部分直接作为《架构设计文档》提交。另外，业界早已存在一些有影响力的多视图方法（例如 RUP 4+1 视图方法），但是作为一线架构师，要有意识地调整、扩充、改进经典方法以符合实践的真正需要。

作为本书主讲的 ADMEMS 方法体系，有哪些具体建议呢？在“Refined Architecture 阶段篇”的后续章节，我们将系统介绍 5 视图方法的实践要领。

第 12 章
Refined Architecture 总论

假设有一座漂亮的大房子，一个人站在房子的前面，一个人站在房子的后面，另外两个人分别站在房子的左右两边。四个人看房子都有不同的视角，四个人都在争论自己看到的那一面是正确的一面。如果运用水平思考，那么这四个人就会绕房子一圈，分别看到房子前后左右四个面。

——爱德华·德·博诺，《六顶思考帽》

总的来说，“架构”一词涵盖了软件架构的所有方面，这些方面紧紧地缠绕在一起，决定如何将之分割成部分和主题显得相当主观。既然如此，就必须引入“架构视点”作为讨论、归档和理解大型系统架构的手段（Generally speaking, the term architecture can be seen as covering all aspects of a software architecture. All its aspects re deeply intertwined, and it is really a subjective decision to split it up in parts and subjects. Having said that, the usefulness of introducing architectural viewpoints is essential as a way of discussing, documenting, and mastering the architecture of large-scale systems.）。

——Peter Herzum，《Business Component Factory》

架构设计是一门解决复杂问题的实践艺术，于是，以分而治之为思想核心的多视图方法必不可少。本章介绍支持细化架构设计的整体思路——多视图方法。

12.1 什么是 Refined Architecture

Refined Architecture 是相对于 Conceptual Architecture 而言的，它们是架构设计的两个层次，分别对应于“概念级”解决方案和“规约级”解决方案（如图 12-1 所示）。须要注意的是，Refined Architecture（细化架构）属于架构设计，不能与 Detailed Design（详细设计）相混淆。

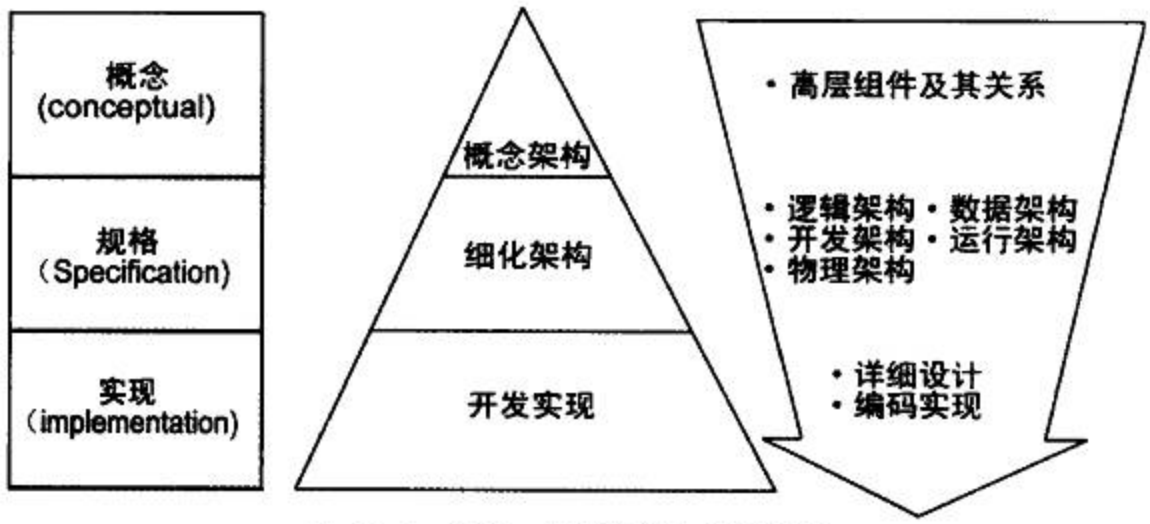


图 12-1　层次：概念架构与细化架构

架构领域最喜欢将建筑设计的多视图方法与软件架构设计的多视图方法做类比，在此就不再赘述，而是举一个更贴近生活的例子：装修的多视图方法。如图 12-2 所示。

- **功能视图**
- **布线视图**

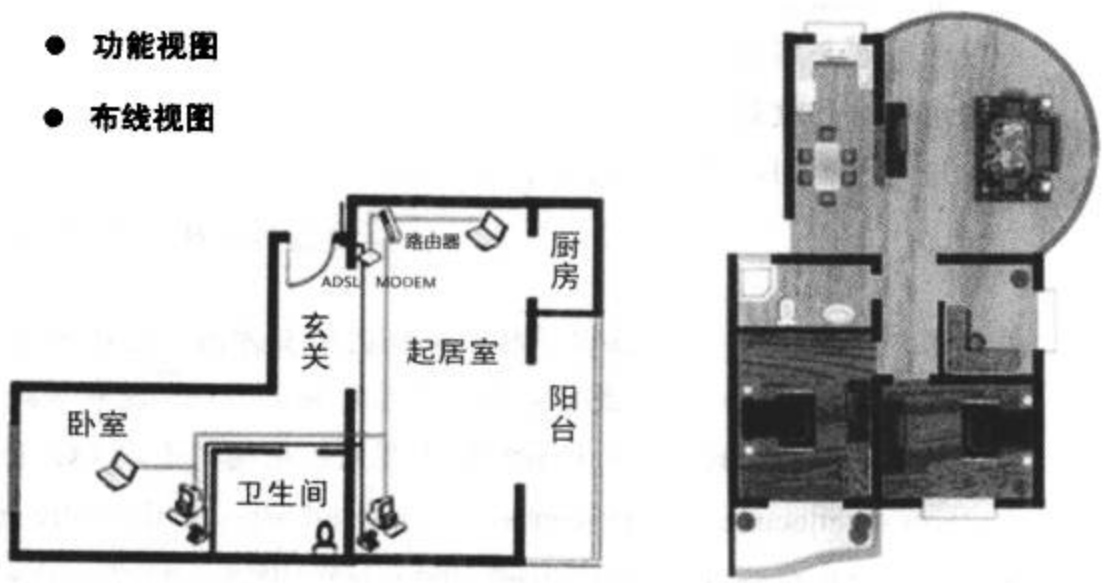


图 12-2　装修时的两视图法

通过运用装修的两视图法，你的装修设计摆脱了在一个图里画来画去（想来想去）的困境。其中，装修设计的功能视图比较多地考虑：

- 家具。
- 家电。
- 灯。
- 窗帘。

而装修设计的布线视图可集中考虑：

- 插座。
- 网线。
- 电话线。
- 有线电视线。

我们还发现，功能视图和布线视图是相互影响的，例如插座不能设计在大衣柜的后面否则无法使用。这与软件架构设计的多视图方法中“兼顾多个视图设计之间的一致性”的要求是神似的，例如架构设计要考虑职责、程序单元、部署节点等要素之间的相互影响。

12.2 实际意义

关于多视图方法的价值，Len Bass 等专家在《软件构架实践（第 2 版）》一书中论述道：

神经科专科医生、整形医生、血液专家和皮肤科医生对人体结构有着不同的视图。眼科医生、心脏病专家和足病医生研究治疗的是身体的某一部分。运动学专家和精神病专家关注的是整个人体行为的不同方面。尽管这些视图是不同的并且具有差异巨大的属性，但它们都具有内在的相关性：它们共同描述了人体的结构。

软件也是如此。现代系统非常复杂，很难一下领会。相反，在任何时刻，我们只能把注意力放在软件系统的一个或几个结构上。为了有意义地传达架构的信息，必须说明此刻正在讨论哪个或哪些结构——即采用的是架构的哪个视图。

所以，多视图方法有两个方面的实际意义：

- 利于思考（因为分而治之的思维方式）。
- 便于交流（因为在一定程度上分离了涉众关注点）。

12.3 业界现状

12.3.1 误认为多视图是 OO 方法分支

提问：Framework 技术是 OO 的分支吗？不是，Framework 本质上和面向对象无关，用 C 语言也可编写 Framework。更切近本质的 Framework 的定义是：可以通过某种回调机制进行扩展的软件系统或子系统的半成品。

的确，OO 方法太流行了，以至于很多技术都“变成”了 OO 的分支。

有同行也常常将多视图方法误认为是 OO 方法的分支。其实，无论是 OO 方法，还是结构化方法，都远未涵盖架构设计的全部。所以，只具有 OO 技能对架构师而言是不够的。

12.3.2 误将“视图”当成“阶段”

对架构设计方法而言，区分阶段和视图的概念是非常重要和必要的。

图 12-3 所示“左边”的观点——概念架构、逻辑架构、物理架构是 3 个不同的层次。其实，

此观点不完全正确，因为逻辑架构和物理架构是架构设计同一阶段中须要同时考虑的两个方面——即二者是两个视图，而非两个阶段。

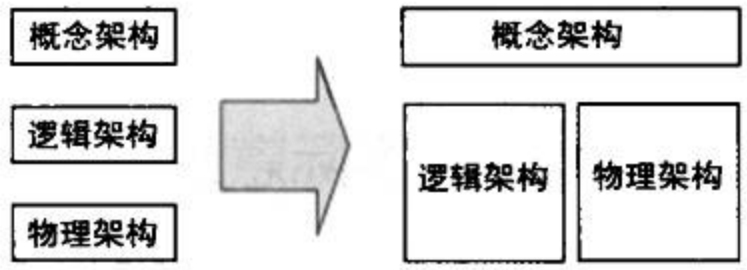


图 12-3　误解与正解

12.3.3　RUP 4+1 视图

在软件架构发展史上，4+1 视图方法具有重大贡献。

1995 年，Philippe Kruchten 发表了题为《The 4+1 View Model of Architecture》的论文，标志着 4+1 视图方法的诞生。后来，Philippe Kruchten 加入 Rational 公司，4+1 视图方法演化为图 12-4 所示的模样。

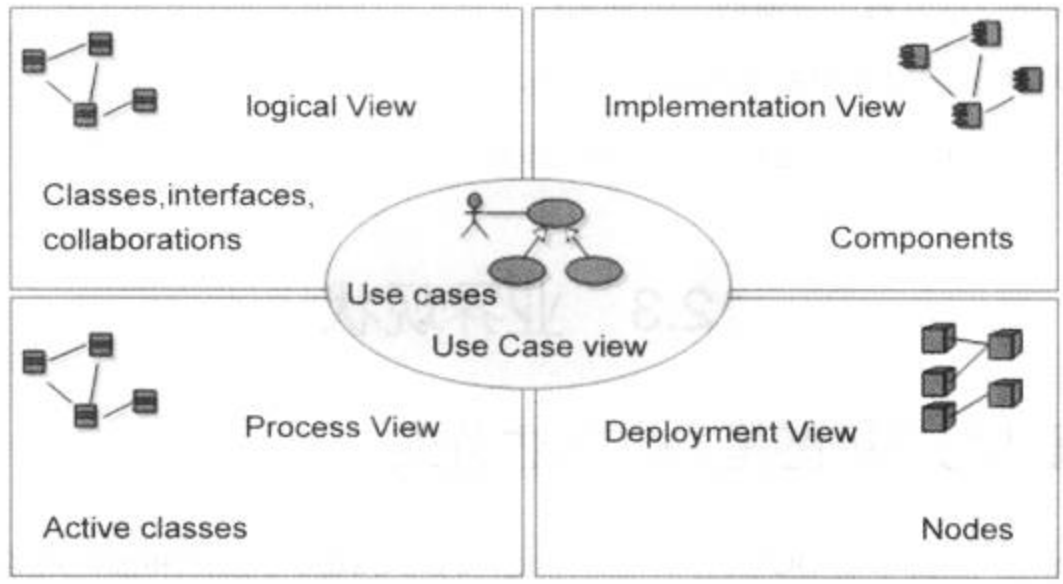


图 12-4　RUP 4+1 视图方法（图片来源：RUP）

RUP 4+1 视图方法有几个重要特点：

- 重视 OO 方法。
- Use Case 驱动。
- 强调模型的重要性。

对应于上述 3 个特点，架构师在实践中应注意：

- OO 可以指导逻辑架构视图的设计，但 OO 方法对物理视图等的设计指导很弱。另一方面，即使是逻辑架构的设计，也未必都是以 OO 方法为指导的。例如，大量嵌入式软件和系统软件仍以 C 语言为主要开发语言，其逻辑架构设计还会以结构化方法为指导。

- 用例不是架构设计本身的工作。4+1 视图中的“4”是架构设计，“+1”是驱动因素。
- 建模切忌穷兵黩武。如果一个模型建立中没有启发思维，首次建立后从不修改，那么就要慎重考虑是不是“过度建模”了。

12.3.4 SEI 3 视图

SEI 的 Len Bass 等专家在《软件构架实践（第 2 版）》中阐述了“3 视图”的观点，他们认为架构设计的工作应包含如图 12-5 所示的 3 类视图（原书的用词是“Software Structures”，即软件结构）：

- 模块视图。此处的元素是模块，它们是实现单元。模块表示一种考虑系统的基于代码的方法。模块被分配功能职责区域。这不怎么强调所开发出来的软件如何在运行时表现自己。模块结构能够回答诸如此类的问题：分配给每个模块的主要功能职责是什么？允许模块使用的其他软件元素是什么？它实际使用的其他软件是什么？什么模块通过泛化或特化（也就是继承）关系与其他模块相关？
- 组件-连接器视图。此处的元素为运行时组件（它们是计算的主要单元）和连接器（它们是组件间通信的工具）。组件—连接器结构回答了诸如此类的问题：什么是主要的执行组件？它们如何交互？什么是主要的共享数据存储？复制系统的哪些部分？数据在系统中经过了哪些地方？系统的哪些部分可以并行运行？在系统执行时，其结构可能会发生怎样的变化？
- 分配视图。分配结构展示了软件元素和创建并执行软件的一个或多个外部环境中的元素之间的关系。它们回答了诸如此类的问题：每个软件元素在什么处理器上执行？在开发、测试和系统构建期间，每个元素都存储在什么文件中？分配给开发小组的软件元素是什么？

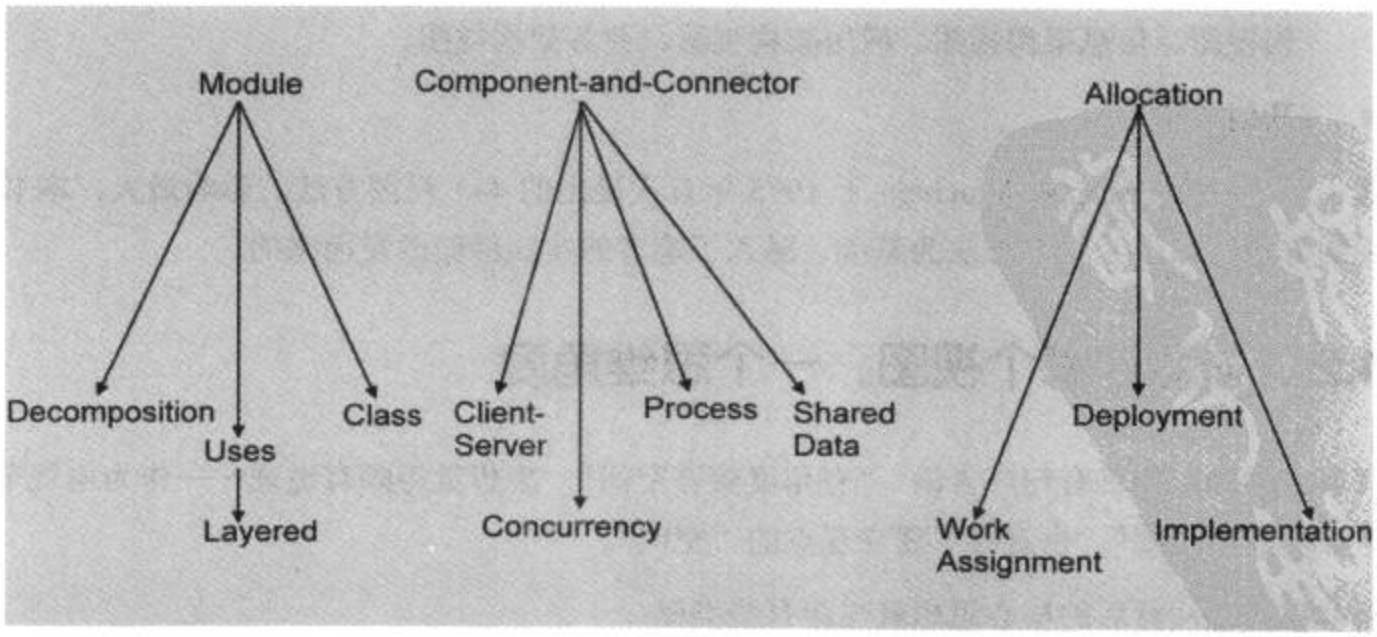


图 12-5 SEI 3 视图方法（图片来源：《软件构架实践（第 2 版）》）

总的来说，SEI 3 视图方法没有 RUP 4+1 视图方法影响大，但也值得架构师研究和体会。例如：

- 映射视图对实践很有启发。以源码为核心的开发单元要分配给开发人员；而目标单元要和物理节点有映射关系，通过安装、部署、烧写等手段完成。
- 组件视图，笔者个人认为有些问题。因为“架构=组件+交互”是业界的基本认识，架构的每个视图都有自己关心的“组件”。

12.4 实践要领

一种优秀的多视图方法，应该能够比较完善地覆盖架构设计的各项工作内容，且将每项工作内容明确地、有理有据地、一目了然地划归到不同架构视图中去。

本节将介绍 5 视图方法的缘起，并用两幅图来说明 5 视图方法的主要思想：错落有致地将众多技术关注点划分成“群落”，“群落”内高聚合，“群落”间松耦合。所以，应用 5 视图方法，有利于架构师设计思维的“有序”展开。

12.4.1 缘起：5 视图方法的提出

多视图方法是业界广泛认同的一种架构设计思路，具体的多视图方法种类繁多：

- SEI 的 3 视图法。涉及视图为：模块视图、组件-连接器视图、分配视图。
- 西门子的 4 视图法。涉及视图为：概念视图、模块视图、代码视图、执行视图。
- RUP 的 4+1 视图法。涉及视图为：用例视图、逻辑视图、开发视图、进程视图、物理视图。
- 联邦企业架构框架（Federal Enterprise Architecture Framework）。涉及视图为：技术架构视图、信息架构视图、应用架构视图、业务架构视图。
- 其他……

其中，无疑由 Philippe Kruchten 于 1995 年首次提出的 4+1 视图方法的影响最大。本书所讲的 5 视图方法是以 4+1 视图方法为基础，融入了笔者的实践经验改良而来的。

12.4.2 总图：每个视图，一个思维角度

《第一财经》栏目有句广告语 “有角度就有空间”，想想觉得颇有道理——作为电视节目，选准了评论的“角度”，也就有了建立观点的“空间”。

而多视图方法背后的核心思想就与此有些类似：

> 从不同角度，规划“分割”与“交互”。
>
> ——*线架构师实践指南*

如图 12-6 所示，5 视图方法包含如下几个视图：

- 逻辑视图。
- 开发视图。
- 运行视图。
- 物理视图。
- 数据视图。

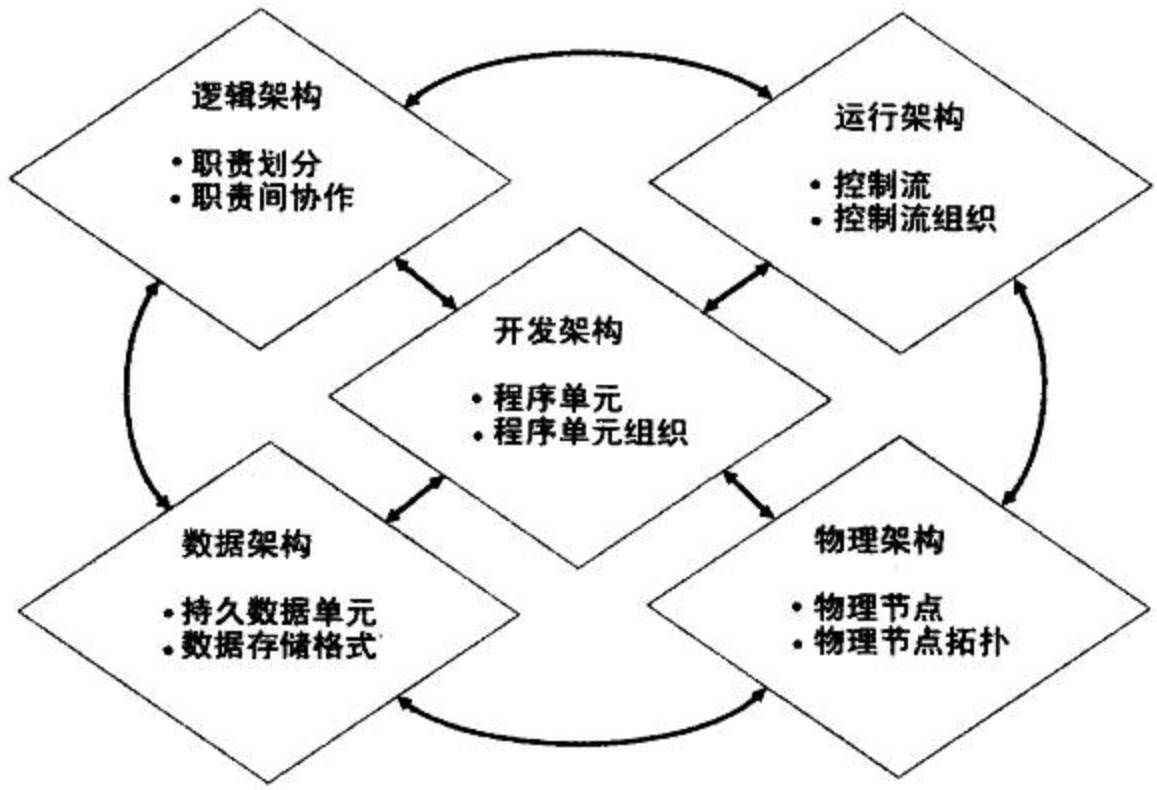


图 12-6 总图：每个视图，一个思维角度

5 个视图各有其“思维立足点”，分别是：

- 职责划分（逻辑视图）。
- 程序单元组织（开发视图）。
- 控制流组织（运行视图）。
- 物理节点安排（物理视图）。
- 持久化设计（数据视图）。

“思考最大的障碍在于混乱”。抓住每个视图的“思维立足点”，5 视图方法就显得“相当清楚”了。

12.4.3 详图：每个视图，一组技术关注点

接下来，看看架构师最常关注的众多技术关注点，如何被 5 视图方法梳理清楚（如图 12-7 所示）。

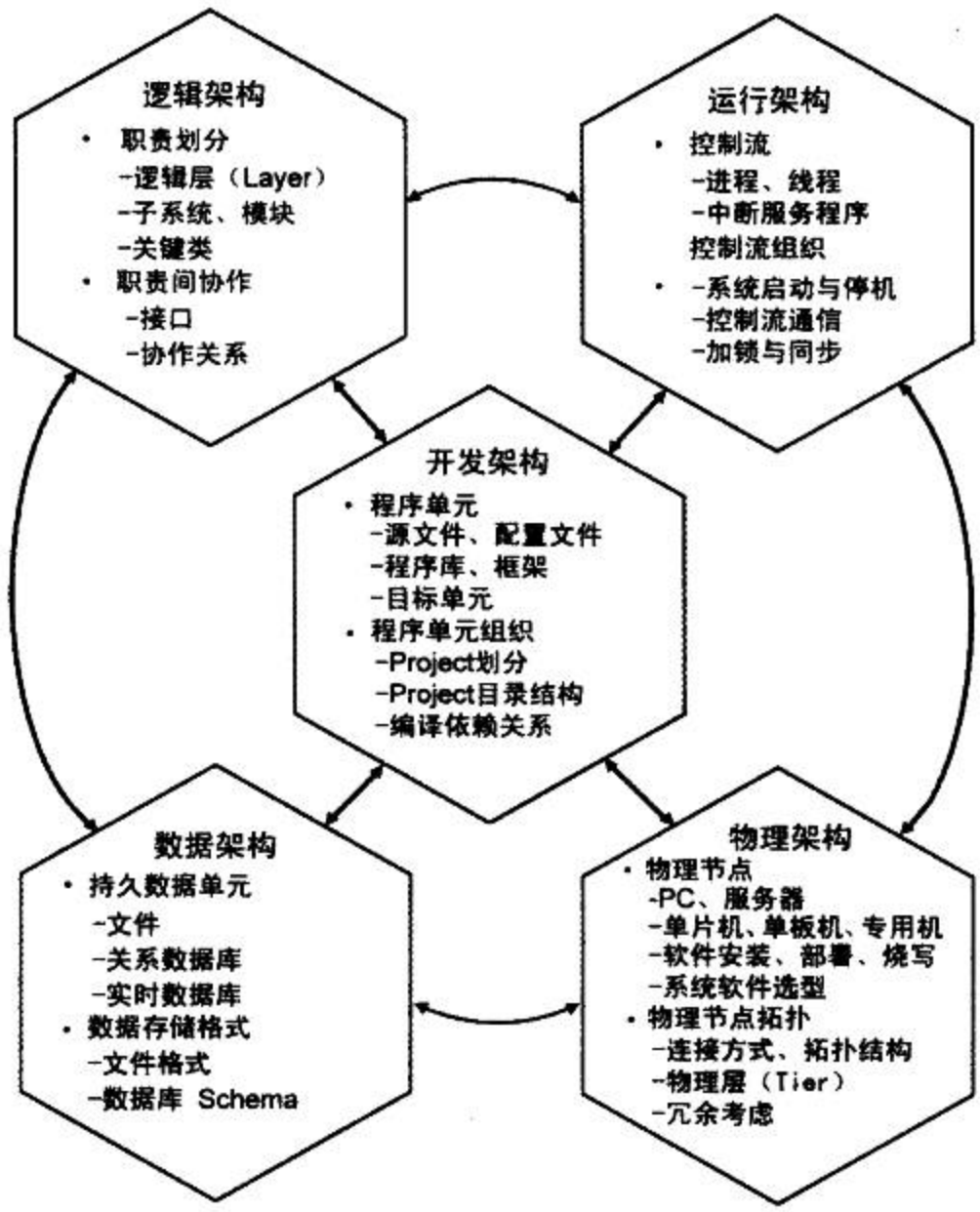


图 12-7 详图：每个视图，一组技术关注点

例如，Layer 是一种大粒度的“职责划分”单位，Layer 的定义属于逻辑架构视图；而 Tier 则属于物理架构视图的考虑范围，它关注硬件部署。

再例如，对于嵌入式系统而言，有经验的架构师能从图 12-7 中“发现”自己的习惯做法：

- 控制流绝不仅仅包含进程和线程，中断服务程序也是一种重要的控制流机制（运行架构视图）。
- 嵌入式应用数据的持久化常常基于文件（而不是数据库表）的概念，最终常写入 Flash（而不是硬盘）中（数据架构视图）。
- ……

如果说每个视图都是一个“语言”的话，那么视图内的这些技术关注点就是语言的“词汇”，正是这些不同的技术关注点支撑起了不同的思维空间。

最后，看似复杂的 5 视图方法其实很简单，因为其每个视图都是从特定角度规划系统的分割与交互，都是（架构的定义）“组件 + 交互”的一种体现。——原来如此，提炼出了“繁”中之“简”，离成功运用这种方法也就不远了！

第 13 章
逻辑架构

有没有一种方法在大产品和小团队之间的缺口上架起一座桥梁呢？答案是肯定的，有！那就是架构。架构最重要的一点，就是它能把难以处理的大问题分解成便于管理的小问题。

——Eric Brechner，《代码之道》

一流是每个程序设计人员向往并为之奋斗却又无法具体说出的、难以达到的境界。一流的软件非常简明。它灵活而清晰，能通过创造性的机制解决复杂的问题，这些机制语义丰富，可应用于其他可能完全无关的问题。一流意味着寻求恰当的抽象，意味着通过新的途径合理利用有限的资源。

—— Grady Booch，《面向对象项目的解决方案》

划分子系统、定义接口……，这些典型工作都属于逻辑架构设计的范畴。

本章阐释 ADMEMS 5 视图方法中逻辑架构视图的设计：

- **先从划分子系统的 3 种必用手段讲起；**
- **随后，纠正“我的接口我做主”这种错误认识，代之以“协作决定接口”的正确理解；**
- **而且，本章将解析逻辑架构设计的整体思维套路，解决一线架构师郁闷已久的“多视图方法只讲做什么、不讲怎么做”的问题；**
- **最后，总结逻辑架构设计的 10 条经验要点。**

13.1 划分子系统的 3 种必用策略

一线架构师最缺的不是理论，也不是技术，而是位于理论和技术之间的“实践策略”和“实践套路”。

就划分子系统这个架构师必做的工作而言，其实践策略可归纳为3种：

- 分层的细化。
- 分区的引入。
- 机制的提取。

13.1.1　分层（Layer）的细化

分层是最常用的架构模式，而笔者进一步认为：在架构设计初期，100%的系统都可以用分层架构，就算随着设计的深入而采用了其他架构模式也未必和分层架构矛盾。

于是，架构师在划分子系统时常受到初期分层方式的影响——实际上，很多一线架构师最熟知、最自然的划分子系统的方式就是：分层的细化。

例如，图13-1说明了基于3层架构进行“分层细化”的一种方式。

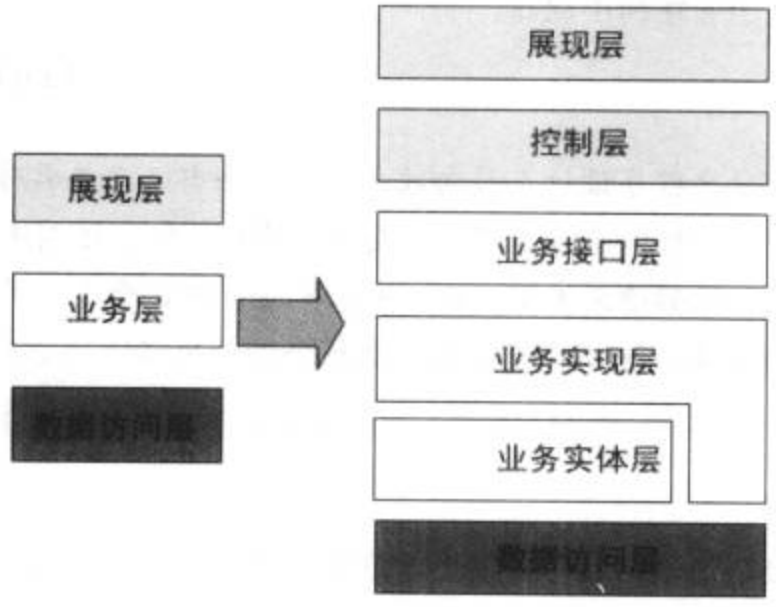


图13-1　基于3层架构进行“分层细化”的一种方式

3层架构或4层架构的“倩影”经常出现在投标时，或者市场彩页中，于是有人戏称之为“市场架构”。的确，直接用3层架构或4层架构来支持团队的并行开发是远远不够的。所以，“分层的细化”是划分子系统的必用策略之一，架构师们不要忘记。

13.1.2　分区（Partition）的引入

序幕才刚刚拉开，划分子系统的工作还远远没有结束。

迭代式开发挺盛行，但所有真正意义上的迭代开发，都必须解决这样一个“困扰”：如果架构设计中只有“层”的概念，以“深度优先”的方式完成一个个具体功能就是不可能的！如图13-2所示，就是一线程序员们经常遇到的烦恼。

架构师：分层架构！
程序员：额的神呀，怎么先开发一个功能？

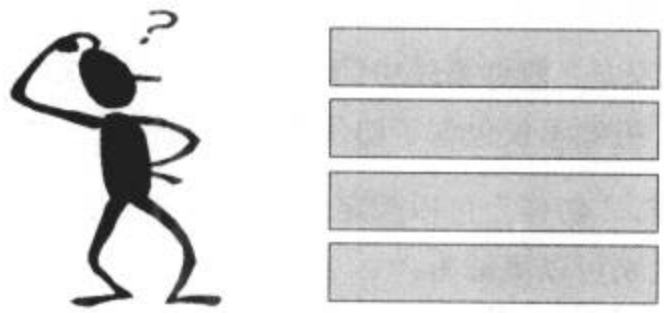


图 13-2 只有分层，如何迭代开发呢？

例如《代码之道》一书中就论及了这一点：

为了得到客户经常性的反馈，快速迭代有个基本前提：开发应该“深度优先”，而不是“广度优先”。

广度优先极端情况下意味着对每一个功能进行定义，然后对每个功能进行设计，接着对每个功能进行编码，最后才对所有功能一起进行测试。而深度优先极端情况下意味着对每个功能完整地进行定义、设计、编码和测试，而只有当这个功能完成了之后，你才能做下一个功能。当然，两个极端都是不好的，但深度优先要好得多。对于大部分团队来说，应该做一个高级的广度设计，然后马上转到深度优先的底层设计和实现上去。

为了支持迭代开发，逻辑架构设计中必须（注意是必须）引入分区。分区是一种单元，它位于某个层的内部，其粒度比层要小。一旦架构师针对每个层进行了分区设计，“深度优先”式的迭代开发就非常自然，图 13-3 说明了这一点。

架构师：分层+分区，必须地！
程序员：明白，让我们开始迭代开发吧。

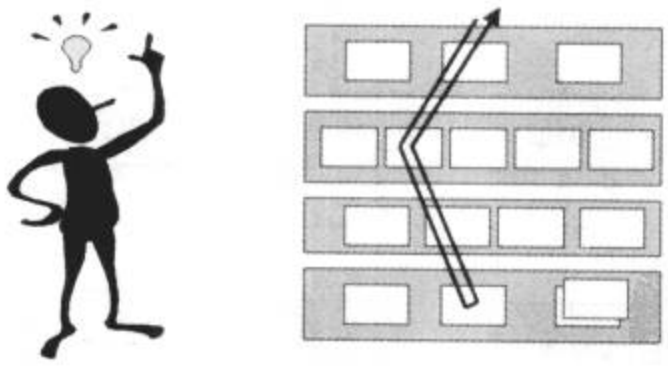


图 13-3 架构中引入分区（Partition），支持迭代开发

架构是迭代开发的基础。架构师若要在“支持迭代”方面不辱使命，必须注重“分区的引入”——这也是划分子系统的必用策略之一。

13.1.3 机制的提取

Grady Booch 在他的著作中指出：

机制才是设计的灵魂所在……否则我们就将不得不面对一群无法相互协作的对象，它们相互

推搡着做自己的事情而毫不关心其他对象。

机制之于设计是如此地重要。那么，什么是机制呢？

本书为“机制”下的定义是：软件系统中的机制，是指预先定义好的、能够完成预期目标的、基于抽象角色的协作方式。机制不仅包含了协作关系，同时也包含了协作流程。

对于面向对象方法而言，“协作”可以被定义为“多个对象为完成某种目标而进行的交互”，而“协作”和“机制”的区别可以概括为：

> 基于接口（和抽象类）的协作是机制，基于具体类的协作则算不上机制。

图 13-4 与图 13-5 显示了协作与机制的不同。

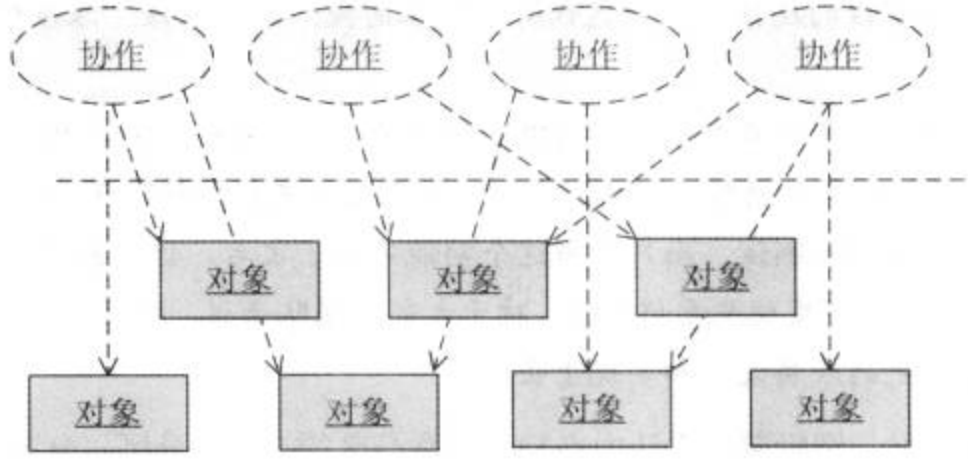


图 13-4　直接组装也称为协作

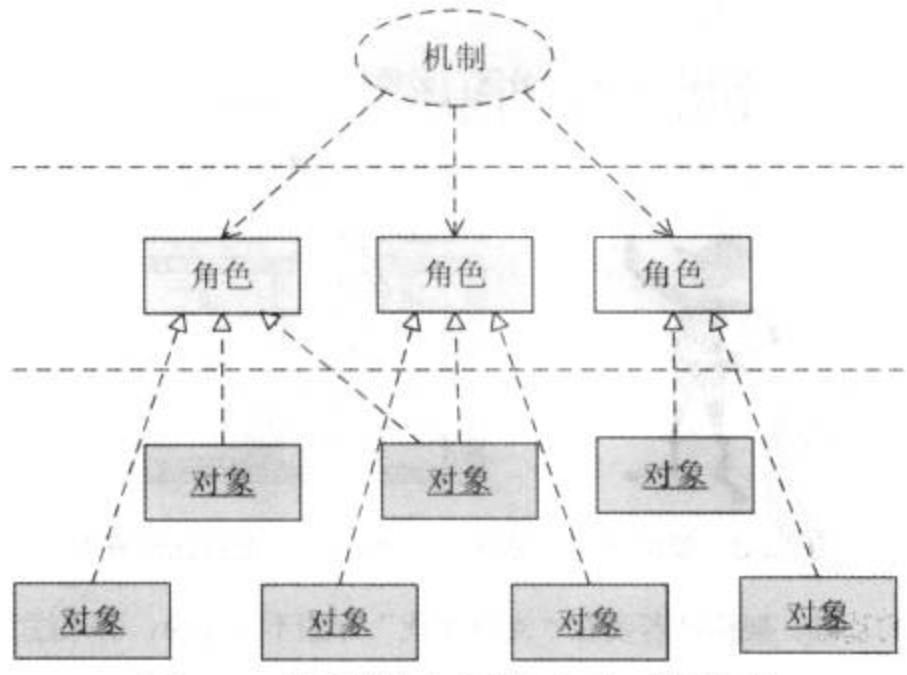


图 13-5　基于抽象角色的协作才可称为机制

对于编程实现而言，在没有提取机制的情况下，机制是一种隐式的重复代码——虽然语句直接比较并不相同，但是很多语句只是引用的变量不同，更重要的是大段的语句块结构完全相同。如果提取了机制，它在编程层面体现为“基于抽象角色（OO 中就是接口）编程的那部分程序”。

对于逻辑架构设计而言，机制是一种特殊的子系统——架构师在划分子系统时不要遗忘这点。

最容易理解的子系统，是通过“直接组装”粒度更小的单元来实现软件“最终功能”的子系统；相比之下，作为子系统的机制并不能“直接实现”软件的“最终功能”。在实现不同的最终功能时，可以重用同一个机制，避免重复进行繁琐的“组装”工作。例如，如图 13-6 所示，网络管理软件中拓扑显示和告警通知都可利用消息机制。

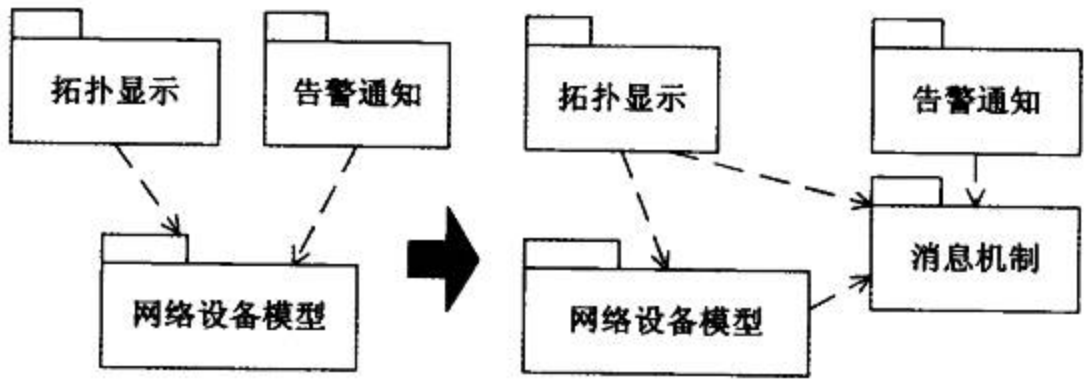


图 13-6 提取通用机制的示例

13.1.4 总结：回顾《软件架构设计》提出的“三维思维”

至此，我们讨论了划分子系统的 3 种手段：分层的细化、分区的引入、机制的提取。通过这 3 种手段的综合运用，就可更理性、更专业地展开逻辑架构的设计，如图 13-7 所示。

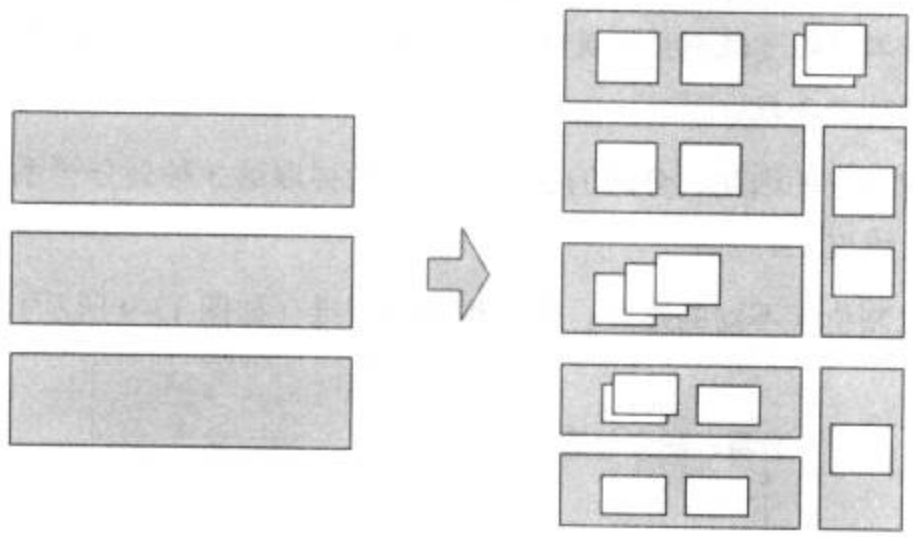


图 13-7 划分子系统必须三管齐下，综合运用 3 种手段

笔者曾在《软件架构设计》一书中阐述：

如何通过关注点分离来达到“系统中的一部分发生了改变，不会影响其他部分”的目标呢？

首先，可以通过职责划分来分离关注点。面向对象设计的关键所在，就是职责的识别和分配。每个功能的完成，都是通过一系列职责组成的“协作链条”完成的；当不同职责被合理分离之后，为了实现新的功能只须构造新的“协作链条”，而需求变更也往往只会影响到少数职责的定义和实现……

其次，可以利用软件系统各部分的通用性不同进行关注点分离。不同的通用程度意味着变化的可能性不同，将通用性不同的部分分离有利于通用部分的重用，也便于对专用部分修改……

另外，还可以先考虑大粒度的子系统，而暂时忽略子系统是如何通过更小粒度的模块和类组

成的……

架构设计关注点分离原理如图 13-8 所示。

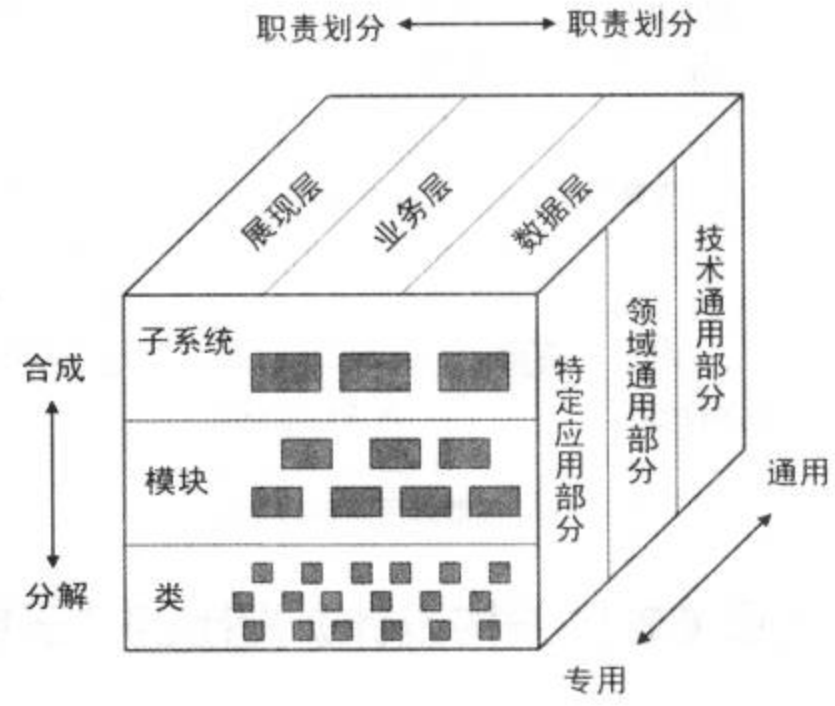


图 13-8　架构设计关注点分离原理示意图

图 13-8 总结了上述的架构设计关注点分离原理。可以说，根据职责分离关注点、根据通用性分离关注点、根据不同粒度级别分离关注点是三种位于不同“维度”的思维方式，所以在实际工作中必须综合运用这些手段。

于是，不难理解分层的细化、分区的引入、机制的提取这 3 种划分子系统手段之间的关系：它们处在思维的 3 个维度上。

首先，分层和机制位于不同的维度：职责维及通用维（如图 13-9 所示）。

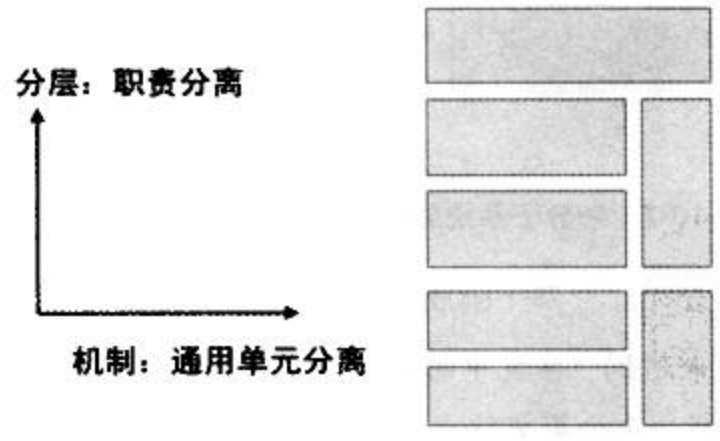


图 13-9　分层和机制位于不同的维度

另外，是否引入分区，设计所“覆盖”的 Scope 是完全相同的。原因是层的粒度较大，而层内部引入的分区的粒度更小，便于组合出一个个功能（支持迭代开发）。这是第三维：粒度（如图 13-10 所示）。

图 13-10 粒度维：分区的引入

看来，分层的细化、分区的引入、机制的提取这3个手段不是相互替代的关系，而是相辅相成的关系。实践中，架构师应三管齐下，综合运用。

13.1.5 探究：划分子系统的4个重要原则

重要的内容就值得多讲几遍，但我会换角度。

下面是分层的细化、分区的引入、机制的提取这3种策略背后的4个通用设计原则：

- 职责不同的单元划归不同子系统。
- 通用性不同的单元划归不同子系统。
- 需要不同开发技能的单元划归不同子系统。
- 兼顾工作量的相对均衡，进一步切分太大的子系统。

如图 13-11 所示，子系统的每种划分策略，都是一到多个原则综合作用的结果。

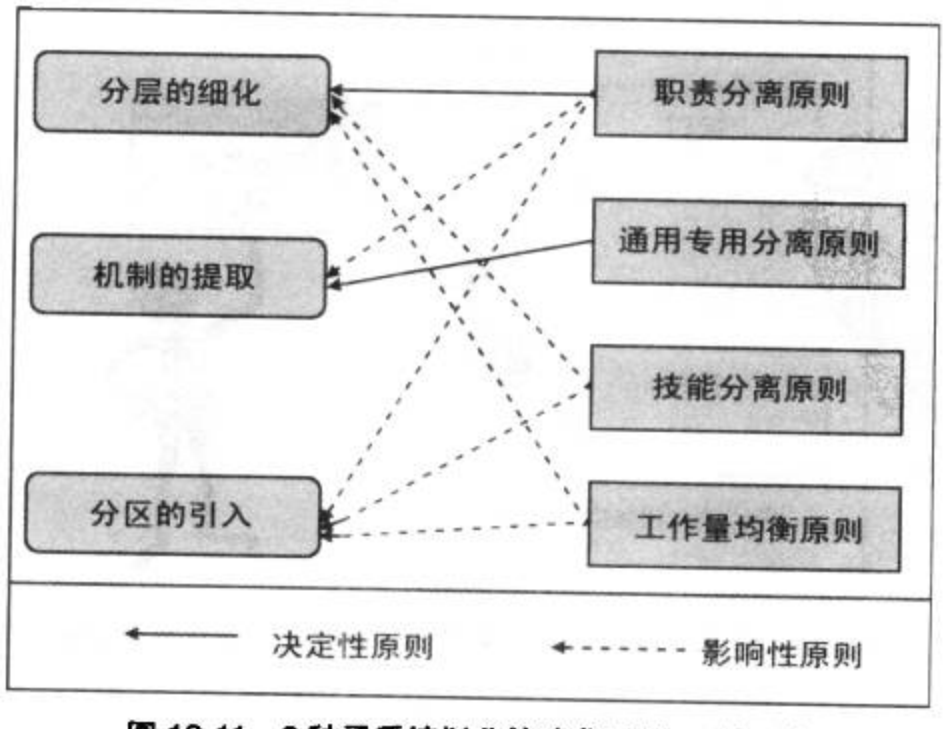


图 13-11 3种子系统划分策略背后的4大原则

13.2　接口设计的事实与谬误

世界是复杂的，很多东西难以直接获得。例如，直接追求幸福，是永远追不到的。（《乐在工作》一书中说幸福是副产品。）

殊不知，合理的接口设计也不是“直接”得到的。

由于面向对象非常强调“自治”，许多人不知不觉地形成了一种错误认识：面向对象推崇“我的接口我做主”。很遗憾，“自治”正确，但“我的接口我做主”这个推断是错误的。

软件世界中本无模块。1968 年，Dijkstra 发表了第 1 篇关于层次结构的论文《The Structure of THE—multiprogramming System》。1972 年，Parnas 发表论文《On the criteria To Be Used in Decomposing System into Modules》论及了模块化和信息隐藏的话题……这些是架构学科开始萌芽的标志。

那么，为什么要对软件进行模块化设计呢？是为了解决复杂性更高的大问题。于是，我们突然领悟：对问题进行分解，分别解决小问题，其实这只是手段。每个架构师应牢记：

> “分”是手段，“合”是目的。不能“合”在一起支持更高层次功能的模块，又有何用呢？

因此，我们必须把模块放在协作的上下文之中进行考虑。架构师设计接口时，要考虑的重点是“为了实现软件系统的一系列功能，这个软件单元要和其他哪些单元如何协作”，总结成一句话就是：

> 协作决定接口。

相反，直接设计接口，是很多“面向接口的”架构依然拙劣的原因之一。类似“我的接口我做主”的观点是错误的（如图 13-12 所示），每个模块或子系统（甚至类）无视协作需要而进行的接口定义很难顺畅地被其他模块或子系统使用。

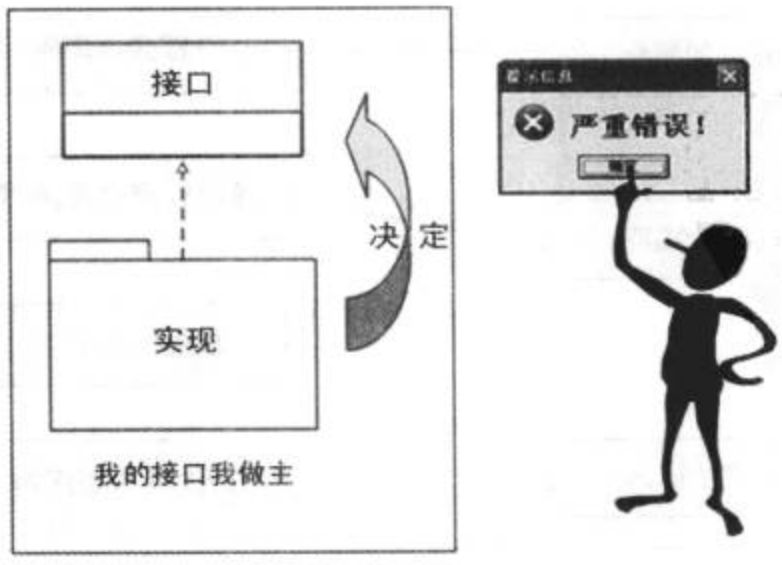


图 13-12　协作决定接口，“我的接口我做主”的观点是错误的

“世界上最短的距离不是直线”，又多了一个例证。

13.3 逻辑架构设计的整体思维套路

13.3.1 整体思路：质疑驱动的逻辑架构设计

要点如下：

- 质疑驱动。
- 结构设计和行为设计相分离。

罗马不是一日建成的。需求对架构设计的“驱动”作用，是伴随着架构师“不断设计中间成果→不断质疑中间成果→不断调整完善细化中间成果”的过程渐进展开的。打个比方，需求就像“缓释胶囊”，功能、质量、约束这 3 类“药物成分”的药力并不是一股脑释放的，而是缓缓释放的——“缓释”的控制者必然是人，是架构师。

请看图 13-13 所示的“药力释放机理”——逻辑架构设计的整体思维套路。

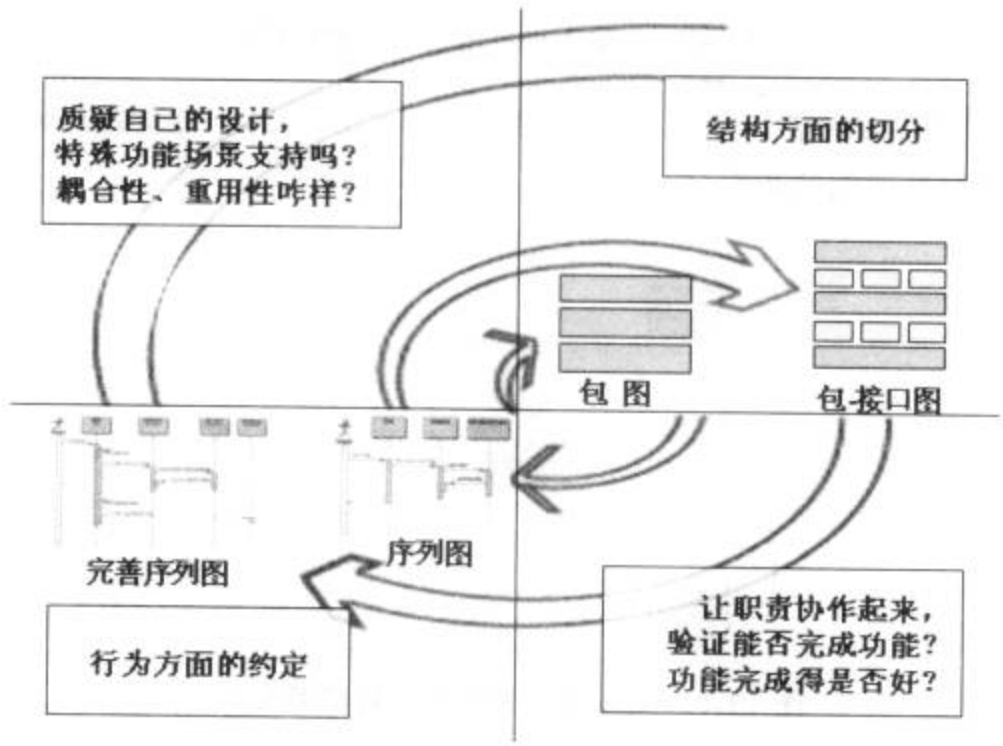


图 13-13 逻辑架构设计的整体思维套路

先考虑结构方面的切分。手段是上面所讲分层的细化、分区的引入、机制的提取。

然后，让切分出的职责协作起来，验证能否完成功能。这个工作，可以借助序列图进行。

此时，结构和行为方面各进行了一定的设计，就应开始质疑自己的设计。架构师要从两个角度质疑：

- 功能方面，特殊的功能支持吗？
- 质量方面，耦合性、重用性、性能等怎么样？

如此循环思维，不断将设计推向深入……其间，会涉及接口的定义，ADMEMS 方法建议用“包—接口图”作为从结构到行为过渡的桥梁，从而识别接口。至于接口的明确定义（接口包含

的方法为何），则要进一步考虑基于职责的具体交互过程。

13.3.2　过程串联：给初学者

第 1 步，根据当前理解切分（如图 13-14 所示）。质疑驱动的逻辑架构设计整体思路，是从运用分层的细化、分区的引入、机制的提取进行子系统划分开始的。

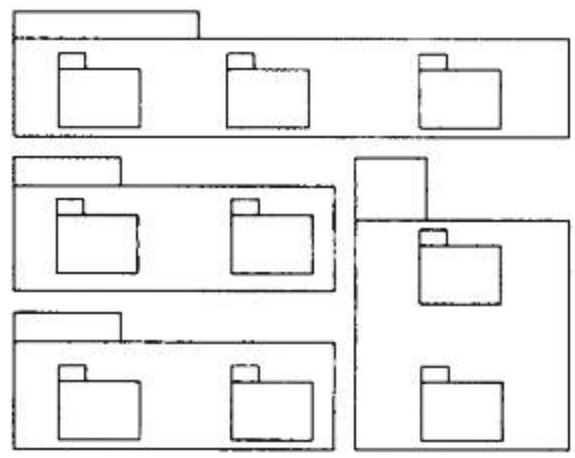


图 13-14　第 1 步：根据当前理解切分

第 2 步，找到某功能的参与单元（如图 13-15 所示）。若找不到或明显缺单元，就可以直接返回第 1 步了，以补充遗漏的职责单元。

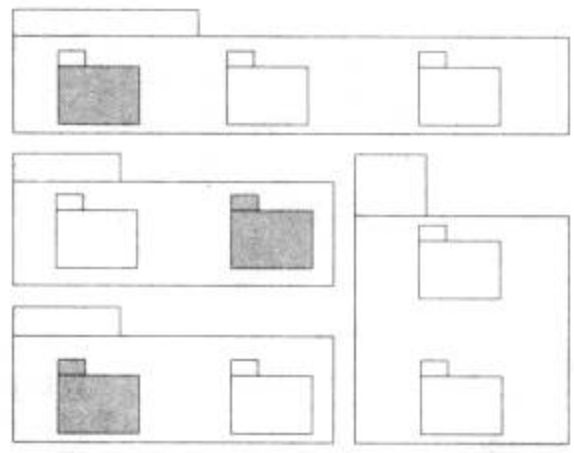


图 13-15　第 2 步：找到某功能的参与单元

第 3 步，让它们协作完成功能（如图 13-16 所示）。研究第 2 步找到的参与单元之间的协作关系，看看能否完成预期功能，完成得怎么样？

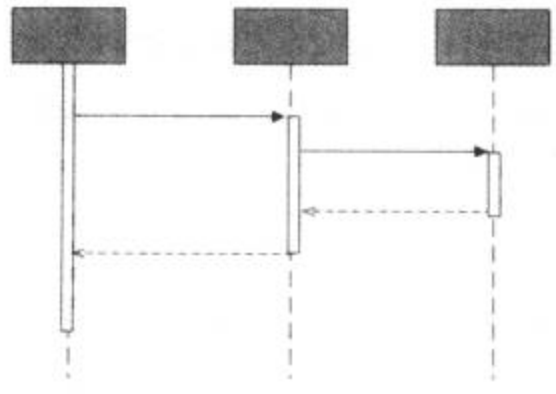


图 13-16　第 3 步：让它们协作完成功能

第 4 步，质疑并推进设计的深入（如图 13-17 所示）。通过质疑“对不对”和“好不好”，可以发现新职责，或者调整协作方式。这意味着，第 1 步的子系统切分方案被调整、被优化……如此循环。

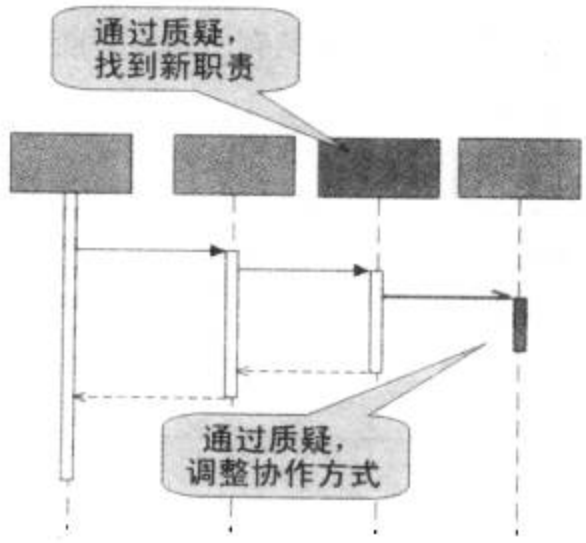


图 13-17 第 4 步：质疑并推进设计的深入

13.3.3 案例示范：自己设计 MyZip

图 13-18 所示为 MyZip 的概念架构设计。它将和需求一起，影响 MyZip 的细化架构设计。

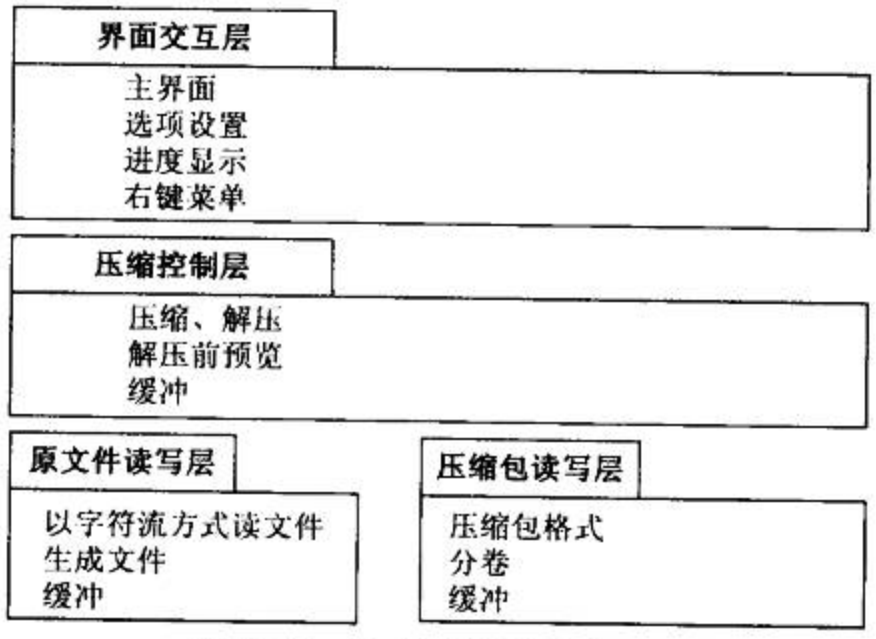


图 13-18 MyZip 的概念架构设计

下面，主要演示如何以质疑驱动的思路，设计 MyZip 的逻辑架构视图。

首先，考虑结构方面的切分，如图 13-19 所示。不难看出，3 种划分子系统的手段都运用了：

- 分层的细化。压缩实现层从原来的压缩控制层中分离出来。回忆本章所讲的“子系统划分策略背后的 4 大原则”。无论是从职责不同的角度，还是从所需技能的角度考虑，两者都应该分离成为单独的“子系统”。
- 分区的引入。界面交互层必须进一步分区，例如：支持右键菜单的“Windows 外壳扩

展”部分被独立。

- 机制的提取。例子是智能缓冲机制，它应该成为一个通用性的基础子系统。同时，为了使它可重用，缓冲区不负责“缓冲区已满”时的具体处理而是回调外部单元进行。再者，为了提高使用友好性，缓冲区具有一定“智能”，它会自动保存溢出的部分，从而简化使用缓冲区的接口。

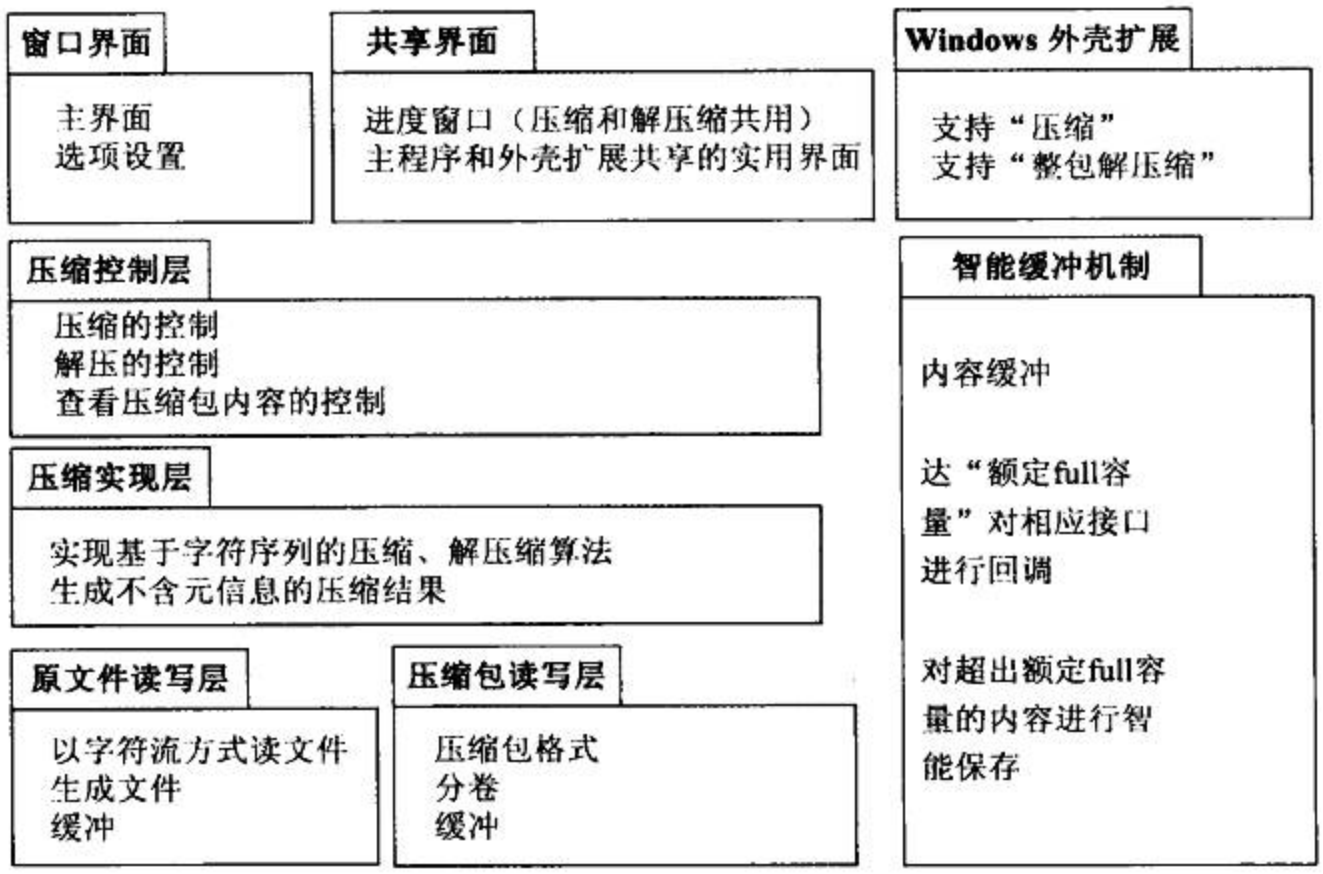


图 13-19 MyZip 逻辑架构设计：首先考虑结构方面的切分

然后，让切分出的职责协作起来，验证能否完成功能。图 13-20 所示的序列图即为一例，初步回答“能协作以支持压缩吗”的问题。请读者看图 13-19，回忆本书提倡的“增量建模”技巧——不要急于“一口吃个胖子”。

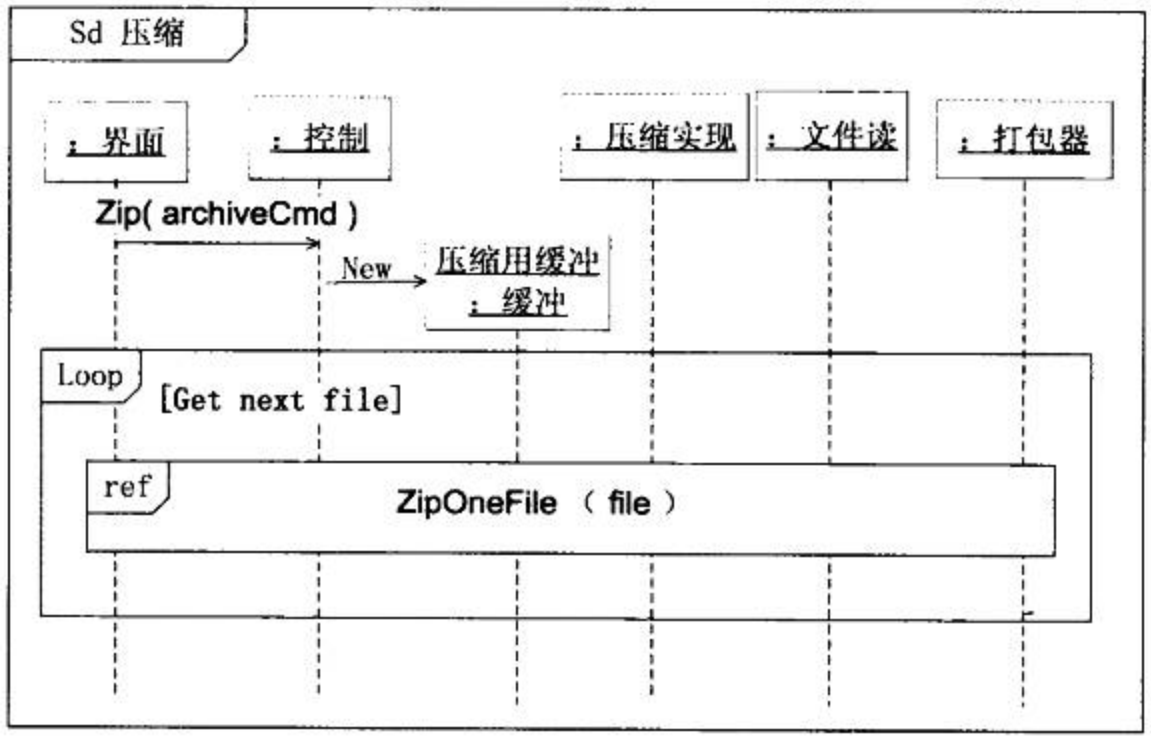


图 13-20 MyZip 逻辑架构设计：协作以验证能否完成“压缩”功能

如此循环，早晚要定义子系统的接口。图 13-21 是包-接口图，帮助架构师明确需要哪些接口（还没有到接口内方法定义一级）。

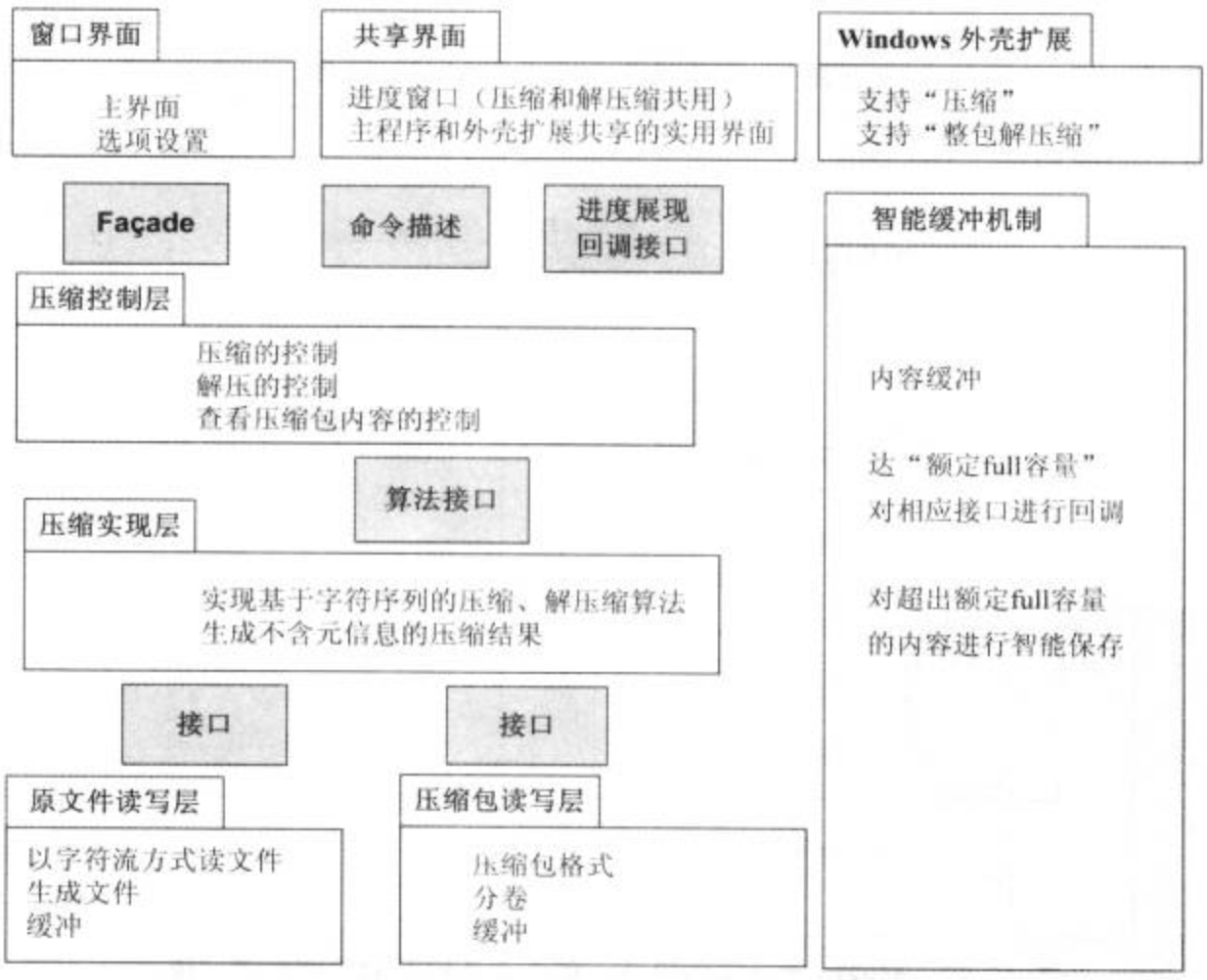


图 13-21 MyZip 逻辑架构设计：包-接口示意图

再次从结构设计跳到行为设计。现在该更明确地考虑压缩了，图 13-22 演示了 ZipOneFile 的设计。同样，遵循“先大局，后局部”的设计原则。具体设计决策是，让“控制”担负 ZipOneFile 的职责，而不是让“压缩实现”来担负——原因是希望“压缩实现”不须感知 File 的概念而能够

更大程度上被重用（例如对数据包而非文件进行压缩）。

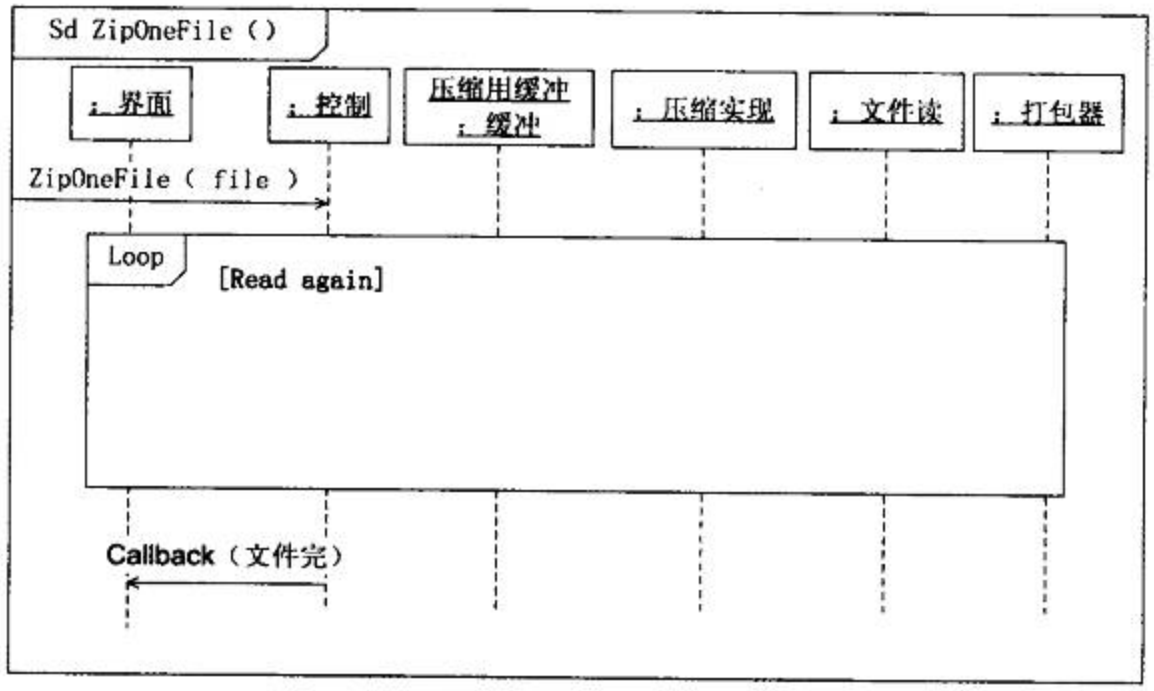


图 13-22　MyZip 逻辑架构设计：每个文件相关的压缩处理

图 13-23 所示的行为设计，离明确接口的方法级定义的目标就不远了……

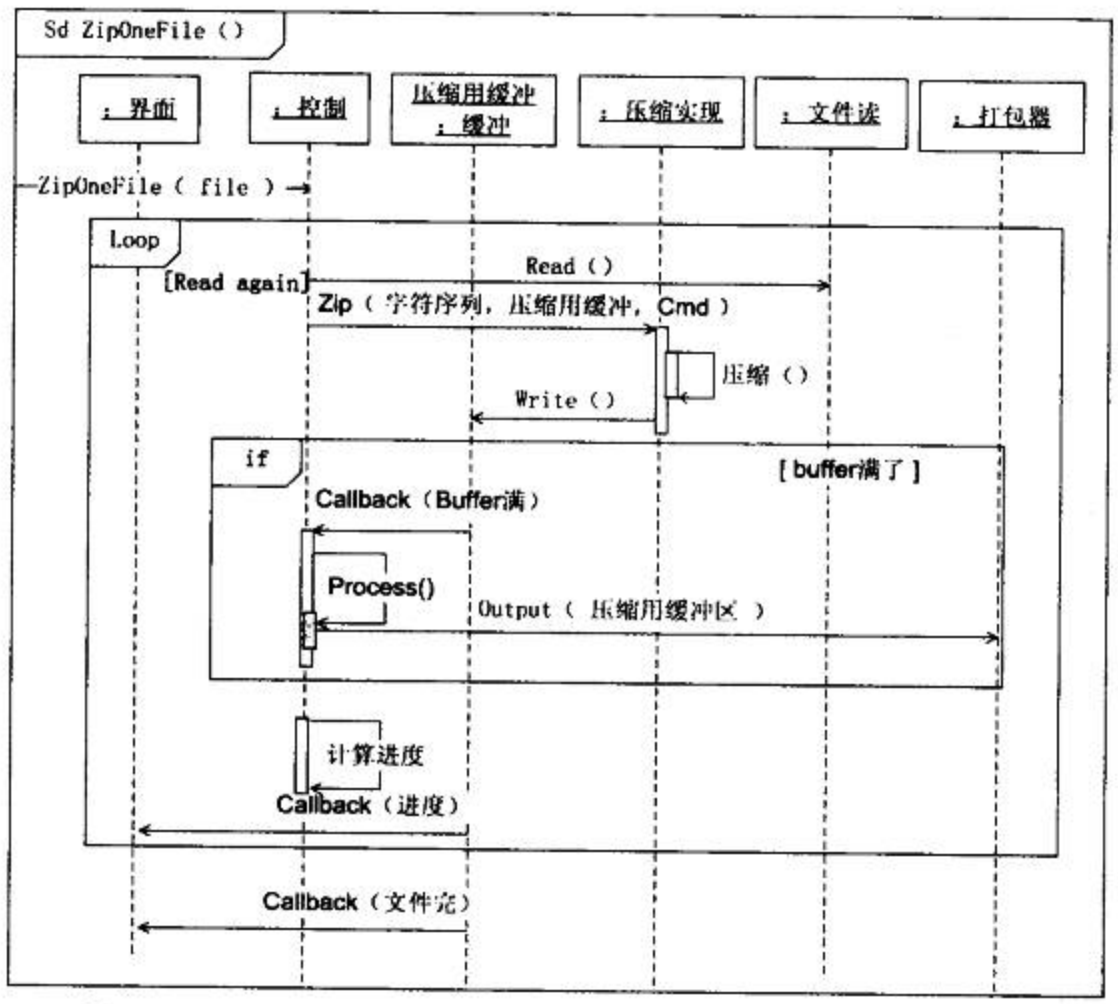


图 13-23　MyZip 逻辑架构设计：每个文件相关的压缩处理（续）

13.4 更多经验总结

13.4.1 逻辑架构设计的 10 条经验要点

图 13-24 归纳了 ADMEMS 方法推荐的逻辑架构设计的 10 条经验要点，其中：如何划分子系统，如何定义接口，如何运用质疑驱动的思维套路等已介绍，其他几点仅在后续小节进行简述。

图 13-24 逻辑架构设计的 10 条经验要点

13.4.2 简述：逻辑架构设计中设计模式应用

设计模式是 Class Level 的设计，它如何用于架构一级的设计呢？

基本观点是：让 Class 和 SubSystem 搭上关系。不难理解，设计模式用于架构设计主要有两种方式：

- 明确子系统内的结构。
- 明确包间的协作关系。

如何做呢？答案是灰盒包图。图 13-25 说明了灰盒包图的意义，它打破了“子系统黑盒”，关心子系统中的关键类，从而可以更到位地说明子系统之间的协作关系，并成为设计模式应用的基础。

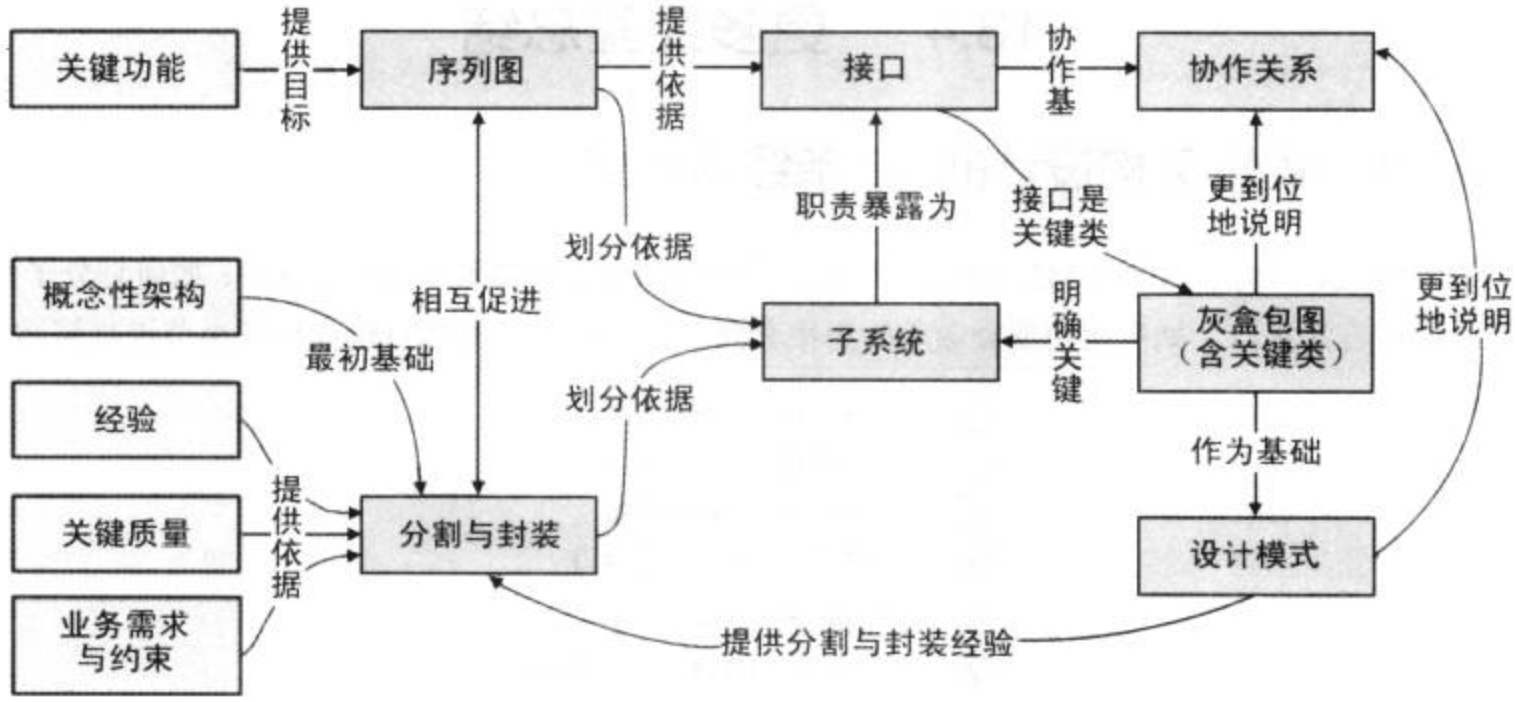


图 13-25　灰盒包图在逻辑架构思维中的“位置”

例如，对比图 13-26 和图 13-27——背景是项目管理系统甘特图展现的问题。后者明确了子系统之间的交互机制，还显式地说明了 Adapter 设计模式的应用——这就是灰盒包图的价值。

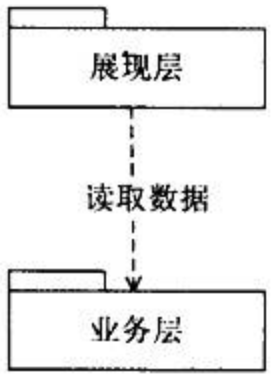


图 13-26　黑盒包图：设计不足

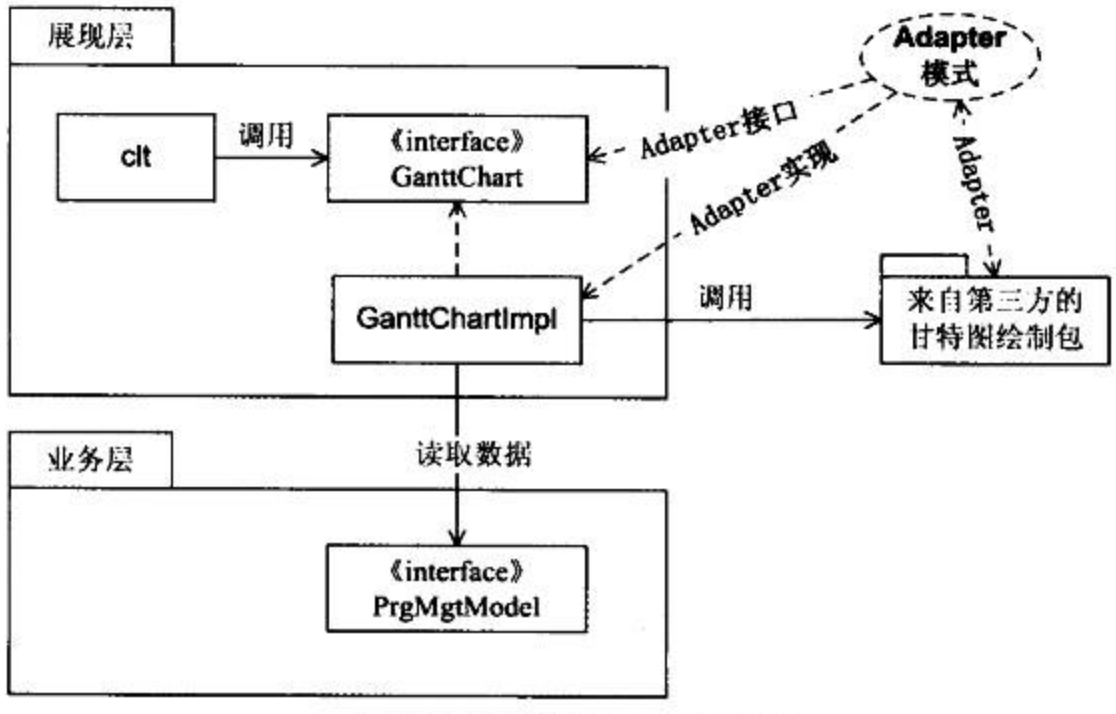


图 13-27　灰盒包图：聪明地折衷

13.4.3 简述：逻辑架构设计的建模支持

工欲善其事，必先利其器。在实践中必须选择最合适的模型，甚至做一些改造工作使 UML 更适合特定的实践目的。例如，灰盒包图就是一种“专门说明重要子系统设计”的 UML 图的应用。

另外，包—接口图是类图的一种特定形式，它包含“包（package）”和“接口（interface）”两种主要元素。这种图（可参考图 13-20）的作用很专一：说明包之间的协作需要哪些接口。逻辑架构设计中，包—接口图式是从结构设计到行为设计的思维桥梁。

最后，本章讲“逻辑架构设计的整体思维套路”时已亮明了观点：逻辑架构的设计，应该使结构设计和行为设计相分离。这样才利于更有效地思维。不信？请看图 13-28 所示的“设计图”（这是很多设计者习惯的思维方式）。思维清楚吗？思维混乱的原因：将结构和行为过多地混在了一起。

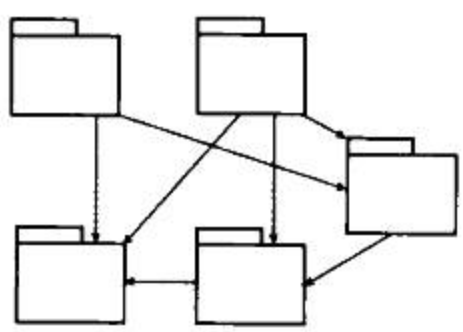


图 13-28 思维混乱的原因：将结构和行为过多地混在了一起

本书推荐用序列图（它较专注于行为设计）辅助逻辑架构设计，尽量不要用协作图（虽然在 UML 1.4 中，它和序列图等价，但从形式上它的“结构气”太重）。

13.5 贯穿案例

下面，继续本书的贯穿案例：PASS 系统的架构设计。首先应注意两点：

第一，细化架构设计的重要“输入”之一是概念架构设计，不应忽视，毕竟细化架构设计是整个架构设计过程中的一个阶段。例如，在第 9 章进行的“基于鲁棒图的初步设计”，以及第 9 章进行的高层分割考虑（如图 13-29 所示）。

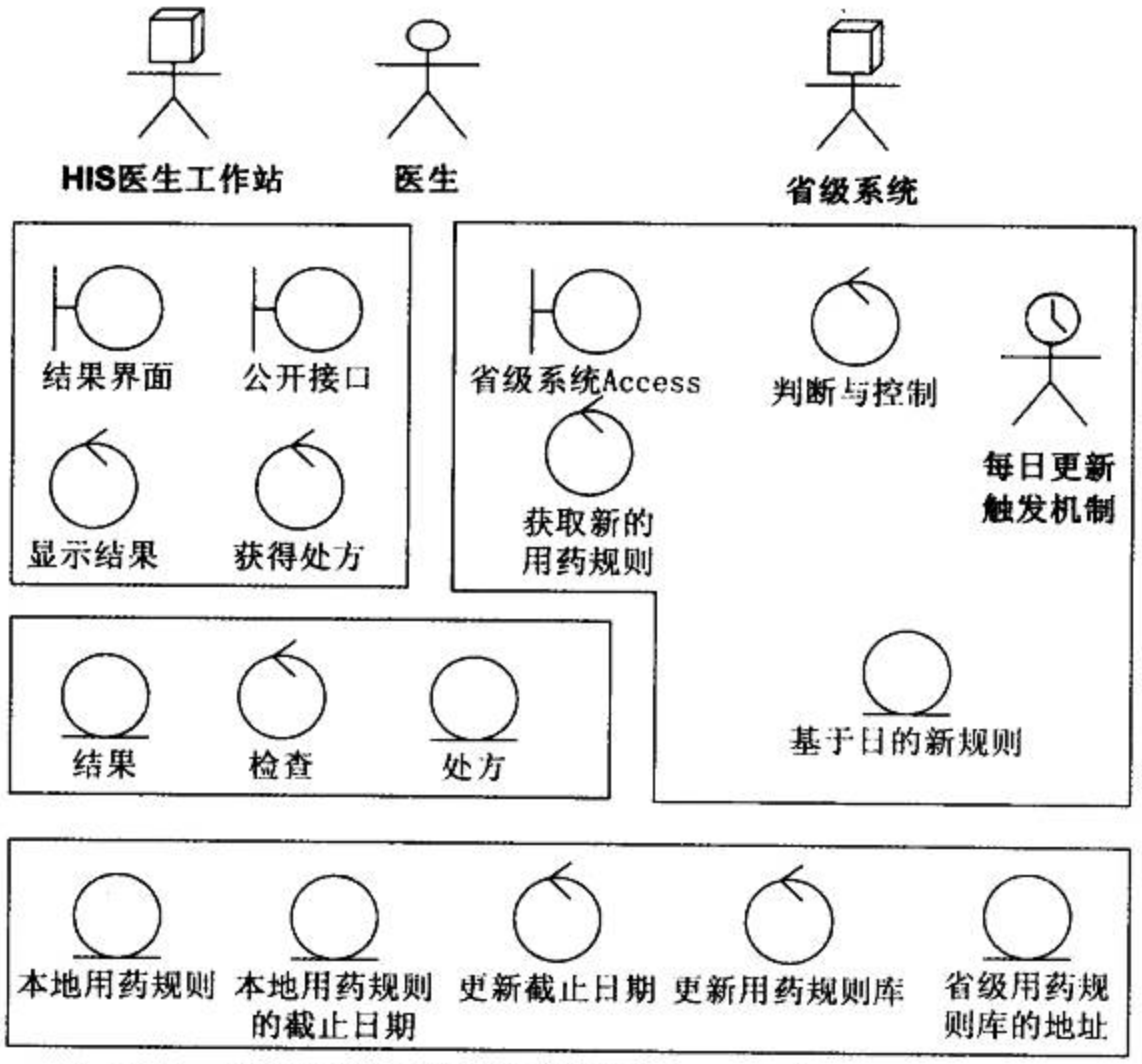


图 13-29 概念架构的思考常为逻辑视图设计提供基础（重复图 9-17）

第二，5 视图方法的运用，总体而言是 5 个视图的设计穿插进行的，对复杂系统而言，根本不可能将逻辑视图设计完全之后再考虑其他视图。而本例的 PASS 系统具有很强的分布特点，所以必然较早地考虑到物理视图对逻辑设计的影响。例如，PASS 服务器作为一个物理架构元素的“节点（Node）”，它之上“跑”的逻辑架构元素“逻辑层（Layer）”有哪些呢？如图 13-30 所示，它包含了业务层、集成层、数据访问层，但未包括展现层程序。

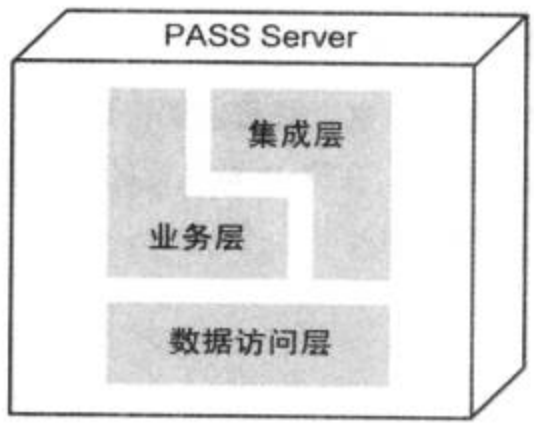


图 13-30 有 3 个 Layer 映射到作为物理节点的 PASS Server

进入细化架构阶段的逻辑架构设计，常以初步设计为基础，借助分层细化、分区引入、机制提取等手段。如图 13-31 所示，对 PASS 服务器软件进行逻辑架构的结构设计。

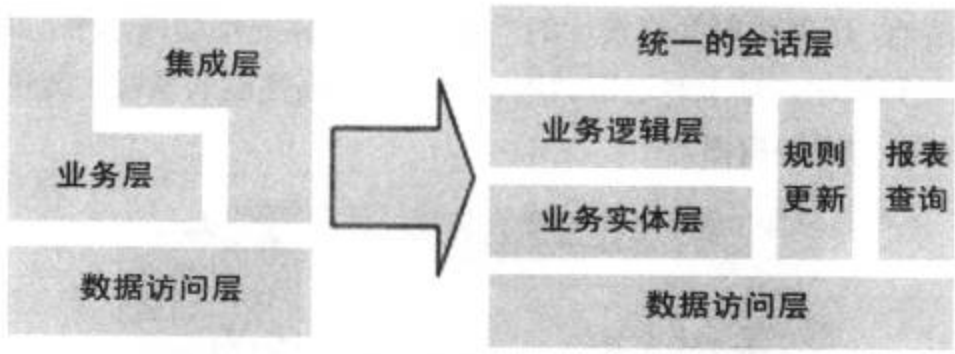


图 13-31 PASS Server 的逻辑架构设计（中间成果）

从结构设计跳到行为设计，常用手段是画序列图。图 13-32 是“实时检查处方功能”的序列图——它处在逻辑架构设计的“螺旋式”整体思维套路的起始循环，是进一步深入设计的基础。

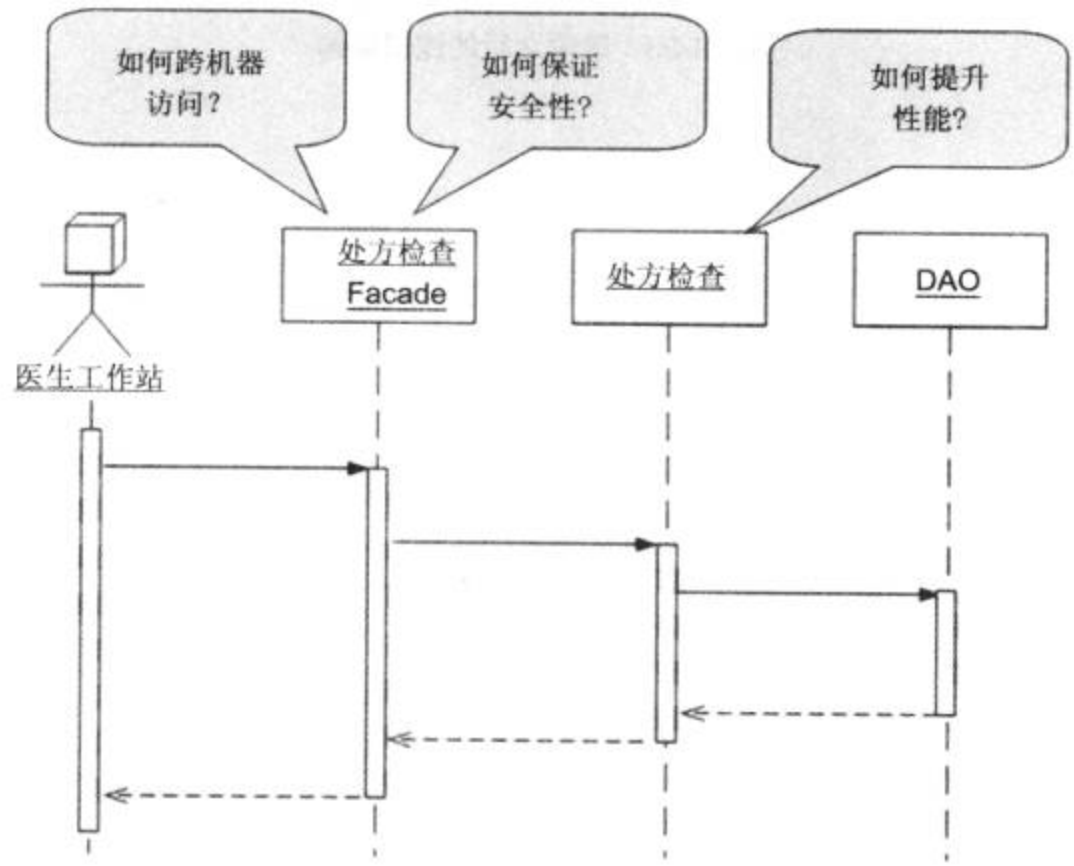


图 13-32 实时检查处方功能：最初的序列图

有了不同职责单元之间具体的协作关系，就可以展开细致的“质疑”了——别忘了，架构设计是质疑驱动的（相关标注同样显示在图 13-32 上）。

- 处方检查服务能被医生工作站访问到吗？毕竟前者位于 PASS 服务器上……于是，设计中要进一步明确“远程调用机制”。
- 这样一个分布式的系统，访问服务之前要经过什么样的验证呢……于是，进一步考虑安全性的支持。
- 不同的医生不停地开处方，处方检查功能会不会很慢？常用药的用药规则应该常驻内存，这样才能提升性能……于是，设计中要进一步明确 Cache 等提升性能的机制。
- ……

于是，自然而然地，沿着逻辑架构设计的“螺旋式”整体思维套路思考，我们就能意识到“结构设计”要继续完善和细化。基于对远程调用、安全性、高性能的质疑，改进“结构设计”后就得到如图 13-33 所示的逻辑架构图。

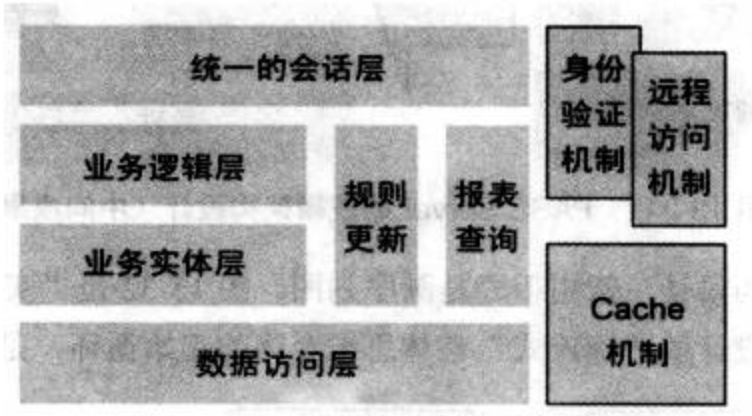


图 13-33　改进之后的逻辑架构

第 14 章
物理架构、运行架构、开发架构

我认识一些架构师，他们的生活都是失控的。因为架构天性范围宽广，涉及的人、工作量都非常多。一些架构师把他们的时间整天整天地花在跟“项目干系人”开会上，然后夜以继日，再搭上周末去做实际的架构工作。

——Eric Brechner，《代码之道》

多重软件架构视图之所以必不可少，是因为各类涉众（用户、客户、开发人员、测试人员、维护人员、内部操作人员、其他人员）须要从各自的角度理解和使用架构。

——Barry Boehm

架构要涵盖的内容和决策太多了，超过了人脑“一蹴而就”的能力范围，因此采用“分而治之”的办法从不同视角分别设计；同时，这样也为对软件架构的理解、交流和归档提供了方便。

本章介绍的物理架构、运行架构、开发架构作为软件架构的不同视图，它们分别关注不同的方面、针对不同的目标和用途。

14.1 为什么需要物理架构设计

硬件强大了，但数据量在增加，计算复杂性也在提高，所以增加硬件未必能解决问题。

相反，计算与计算往往不是孤立的，它们之间存在着复杂的“生产者—消费者”关系，所以软件的实际服务能力不仅受到“硬件资源”的制约，也受到“数据短缺”和“数据争用”的制约。每个架构师都应该懂得：

> 增加硬件 ≠ 增加计算能力 ≠ 软件的实际服务能力增强

多视图方法中，物理架构视图着重考虑运行软件的计算机、网络、硬件设施等情况，还包括如何将软件包部署（如果是嵌入式系统则是烧写）到这些硬件资源上，以及它们运行时的配置情况。另外，物理架构还要考虑软件系统和包括硬件在内的整个 IT 系统之间是如何相互影响的，由于一部分运行时质量属性需要硬件或网络的支持，所以物理架构必须关注如何配置硬件和网络来满足软件系统的可靠性、可伸缩性、持续可用性、性能、安全性等方面的要求。

14.2　物理架构设计的工作内容

物理架构设计主要有 3 项任务（图 14-1 为从“架构=元素+交互”角度的总结）：

- 硬件选择与物理拓扑；
- 软件到硬件的映射关系；
- 方案的优化。

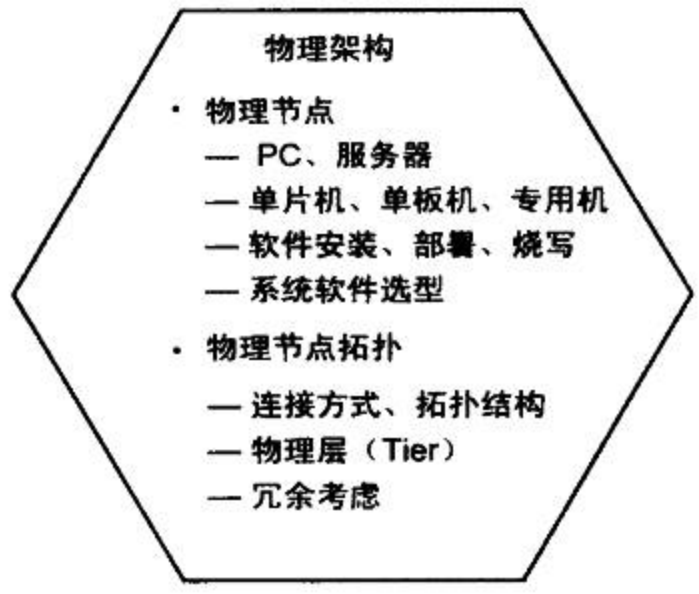


图 14-1　物理架构的设计内容

14.3　探究：物理架构的设计思维

相对于逻辑架构设计而言，物理架构视图的设计是不是就乏善可陈呢？不。一线架构师最缺的是物理架构的设计思维。

从设计结果层面，决策无非围绕物理节点、网络、软件单元、数据单元等内容展开。

但是，思维当中经历了哪些思考、判断和权衡呢？

从最终目标层面，决策要兼顾多方涉众的不同利益，可从“攻”与“守”两个方面理解：

- 高性能（攻）。
- 持续可用性（攻）。

- 可伸缩性（攻）。
- 经济性（守）。
- 技术可行性（守）。
- 易维护性（守）。
- ……

从思维要点层面，“开销”和“争用”是核心。架构师正是通过“降低开销”、“避免争用”来实现高性能、高可伸缩性等最终目标的：

- 如何降低物理节点“内”的计算开销？
- 如何降低物理节点“间”的通信开销？
- 如何避免物理节点“内”CPU、内存、硬盘等资源的争用？
- 如何避免物理节点“间”网络的带宽资源冲突？

至此，我们了解了物理架构设计的理性思维框架，如图 14-2 所示。

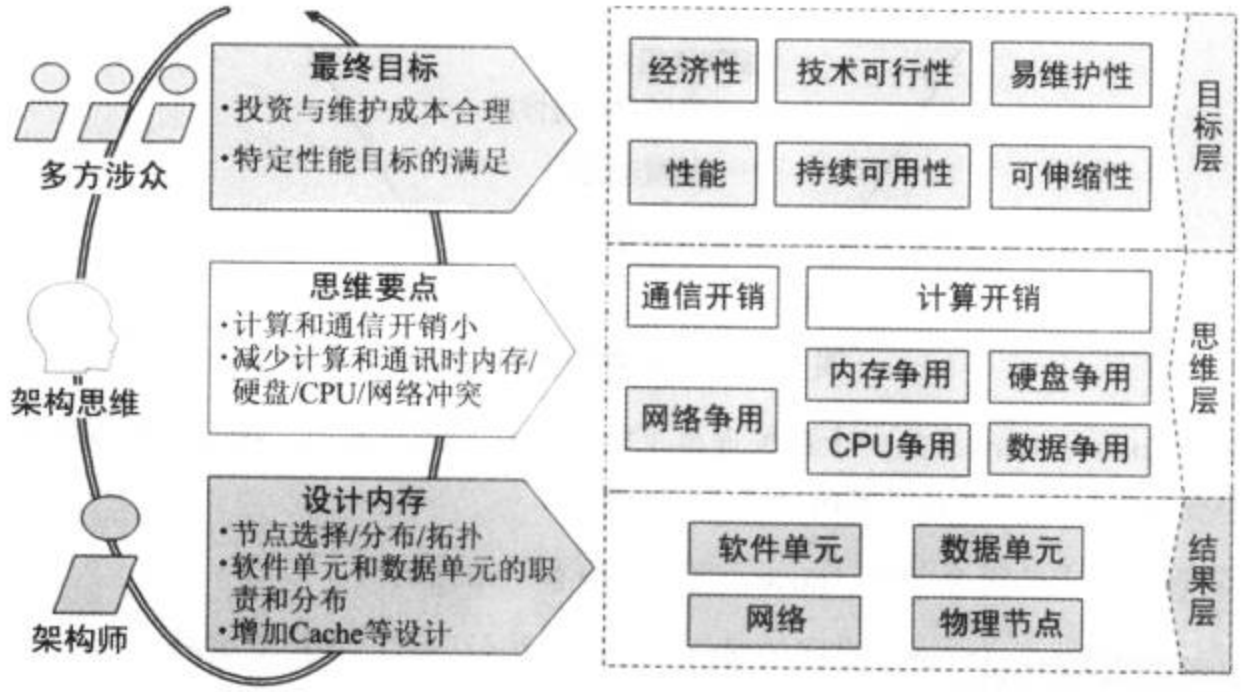


图 14-2 物理架构设计的理性思维框架

14.4 为什么需要运行架构设计

如果系统中，并不准备引入任何并行或并发处理，并且系统也没有基于 SDK、API 等基础软件进行定制开发，那就不须要设计运行架构的设计——这相当于将 5 视图方法剪裁为 4 个视图。

相反，很多系统为了应对复杂的业务逻辑或复杂的互操作逻辑（含硬件交互），或者为了优化关键资源使用效率，而必须借助多条控制流并行或并发执行时，就须要设计运行架构。

14.5　运行架构设计的工作内容

14.5.1　工作内容

根据具体情况不同，运行架构设计可能包括下列工作内容（如图 14-3 所示）。

- 确定引入哪些控制流。
- 确定每条控制流的任务。
- 处理相关问题：控制流的创建、销毁、通信机制等。
- 进一步考虑：控制流之间的同步关系，若有资源争用还要引入加锁机制。

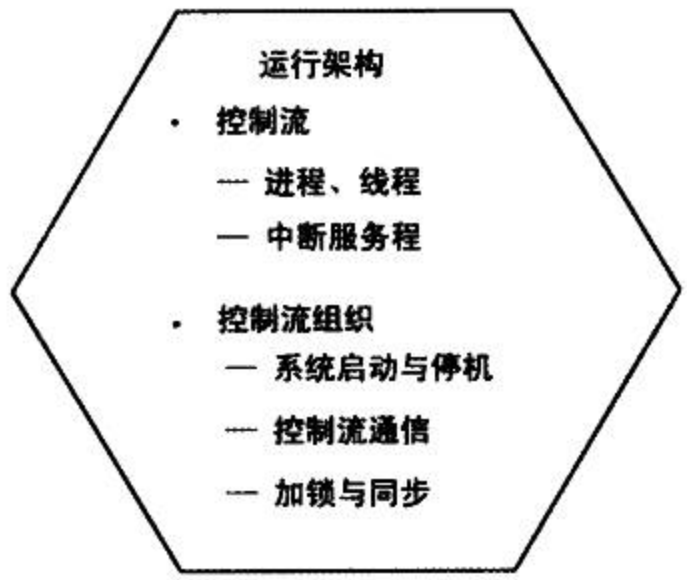


图 14-3　运行架构的设计内容

控制流（Control Flow）是一个在处理机上顺序执行的动作系列。确定引入哪些控制流，并没有固定不变的套路，但有几点考虑是必不可少的：

- 物理架构中的每个节点（node）之上至少有一条控制流。
- 为了实现节点（node）之间的通信，通常做法是引入一条控制流来专门负责。
- 节点（node）是具有主动行为的设备，为其引入专门的控制流（例如中断服务程序）。
- 在需求一级的描述中（例如用例规约中）就是并行或并发的，引入多条控制流。
- 来自用户或外部系统的并发访问，常要求后端服务支持多控制流。
- 如果控制流关系复杂，可以考虑引入对其他控制流进行协调的控制流。

一旦系统中存在不止一条控制流，就产生了附加的工作量，例如控制流的创建、控制流的销毁、建立共享内存或消息等不同控制流之间的通信机制。

14.5.2 控制流图是关键

如图 14-4 所示，控制流图显示了系统中不同控制流之间的关系。控制流的起点是“主动单元”，它会调用其他“被动单元”……如此层层调用，就形成一个控制流。明确了系统中所有的主动单元，就抓住了每个控制流的源头，从而可以把并发执行的所有控制流梳理清楚。

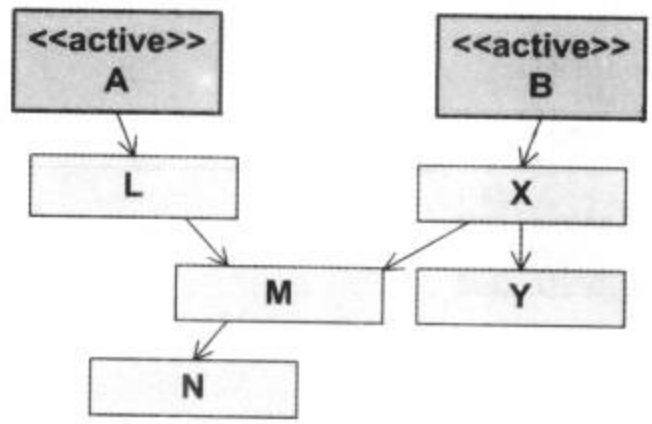


图 14-4 控制流图

运行架构设计的工作看似多而杂，但其实只要把握“控制流图”，就能够提纲挈领地开展其他相关设计。例如，以图 14-4 为例，M 和 N 两个模块可能须要加锁，因为 M 模块的“入度”等于 2，而 N 是 M 的下游模块。

14.6 实现控制流的3种常见手段

在实践中，最常用于实现控制流的手段有 3 种：

- 进程。
- 线程。
- 中断服务程序。

进程（Process）是重量级控制流，既是处理机资源的分配单位，又是其他计算机资源的分配单位。

线程（Thread）是轻量级控制流，仅仅是处理机资源的分配单位。一个进程内可以包含多个线程，后者共享前者的资源；但处理机资源例外，线程是独立的处理机资源的分配单位。

实际上，中断服务程序（Interrupt Service Routine，ISR）也是常见的控制流实现机制。当没有 OS 的支持却要实现并发时，它非常必要。

例如，如图 14-5 所示的多条控制流设计用到了线程，以及中断服务程序的技术（背景为设备调试系统）：

- 应用层中的线程代表主程序的运行，它直接利用了 MFC 的主窗口线程。无论是用户交互，还是串口的数据到达，均采取异步事件的方式处理，杜绝了任何“忙等待”无谓的

耗时，也缩短了系统响应时间。

- 协议层有独立的线程控制着“上上下下”的数据，并设置了数据缓冲区，使数据的接收和数据的处理相对独立，从而使数据接收不因暂时的处理忙碌而停滞，增加了系统吞吐量。
- 硬件控制层的设计中，分别通过时钟中断和 RS232 口中断来激发相应的处理逻辑，达到轮询和收发数据的目的。

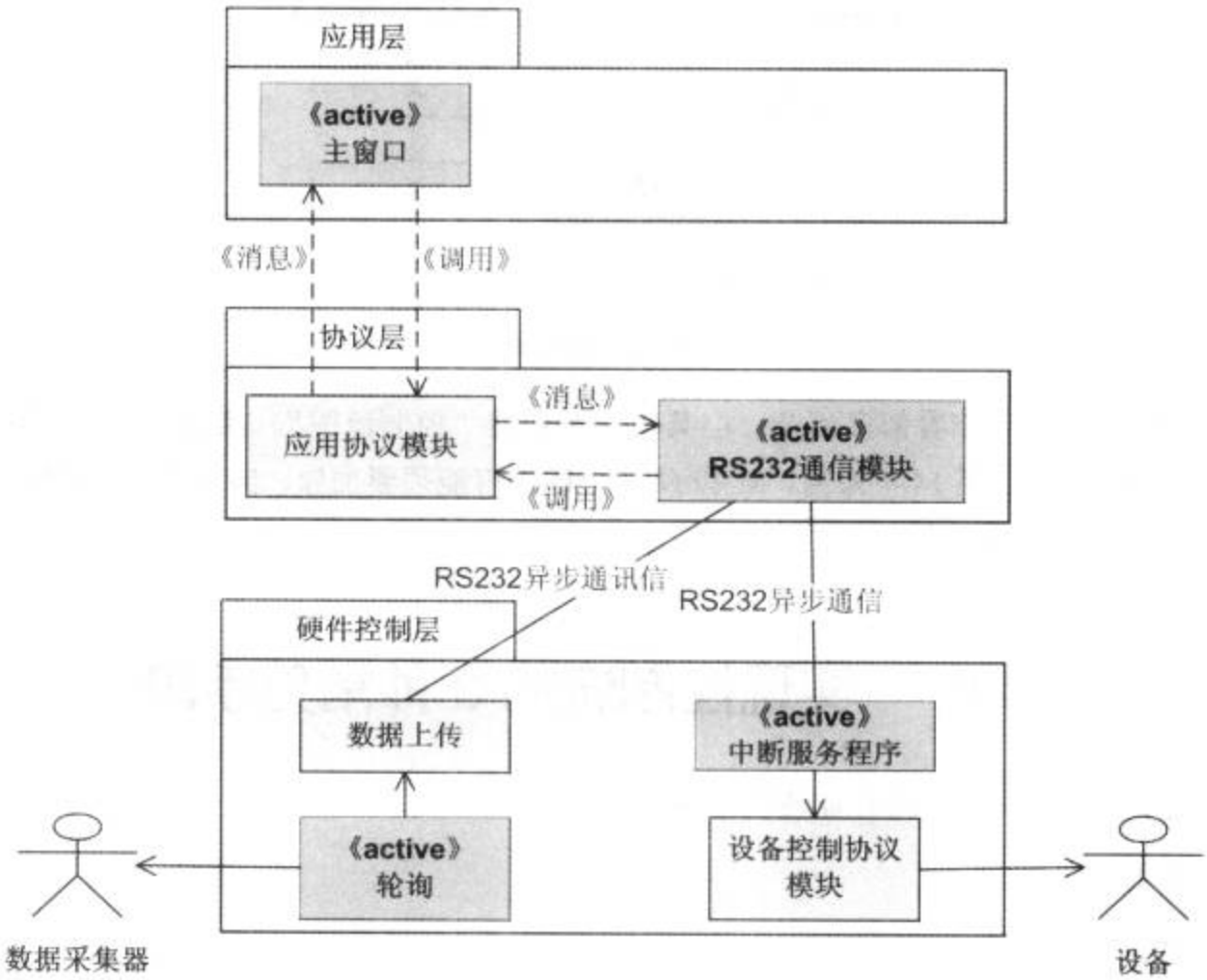


图 14-5　设备调试系统：包含线程、中断服务程序等控制流技术

14.7　为什么开发架构是必须的

以前，很多企业不重视架构；现在，重视架构的企业远比不重视架构的企业多。但是，许多企业发现一个棘手的问题：开发人员不按照架构进行详细设计和编程。

大家一起来思考如图 14-6 所示的练习题。

互动问答

问题：许多公司困扰于：开发人员不按照架构进行详细设计和编程。如何让开发人员更“拥护”架构？

A：在架构设计中重视“开发架构视图”，让开发人员看到他最关心的“程序单元”、“源代码目录结构”等概念

B：架构设计不可“高来高去”，能支持并行的详细设计是“架构设计进行到什么程度”的标志

C：应令 HR 对开发人员批评、教育

D：编程一线的程序经理参与架构设计

图 14-6 为什么开发人员不遵守架构？

此题的答案是：A、B、D。

一句话，架构师在抱怨研发管理、职位权力之前，还须自查！

首先，最基本的一点，架构师必须重视开发架构视图。并行开发所需的“程序单元”、“源码目录结构”等概念，是不同程序团队开展具体工作的基础。如果程序员们总是不能从架构中看到上述内容，就会认为架构是一类“高来高去”的概念，自然不会有积极态度。（A 正确。）

另外，能不能更具体地“定义”架构设计应该达到的程度呢？答案是：能支持并行的详细设计。所以，架构师投标成功之后，切不可将投标中演示的“市场架构”直接作为架构设计的全部。因为这意味着很多影响全局的设计决策被“漏”到了后边，最终到大规模并行开发阶段才发现，造成“程序员碰头儿临时决定”的情况大量出现，必然导致软件质量下降甚至项目失败。（B 正确。）

当然，有能力的架构师，再加上聪明的管理策略就更好。既然每个程序经理都必须深入理解架构，那何不让他们参与到架构实践的工作中来，免去了大量“单纯的架构交流”的工作量。更不必说，“了解产生爱”（程序人员不了解你的架构又如何喜欢你的架构）和“成就感”的心理因素会让程序经理支持架构设计方案。（D 正确。）

14.8 开发架构设计的工作内容

一般而言，开发架构的设计应完成下列工作（如图 14-7 所示）：

- 将“逻辑职责”映射为“程序单元”。
  - 要自主编写的源程序。
  - 可重用的库、框架。
  - 其他方式（如 Shell 脚本、平台支持下的配置文件）。

- 开发技术选型。
  - 开发语言。
  - 开发工具。
- “程序单元”间关系。
  - Project 划分（可选）。
  - Project 目录结构。
  - 编译依赖关系。

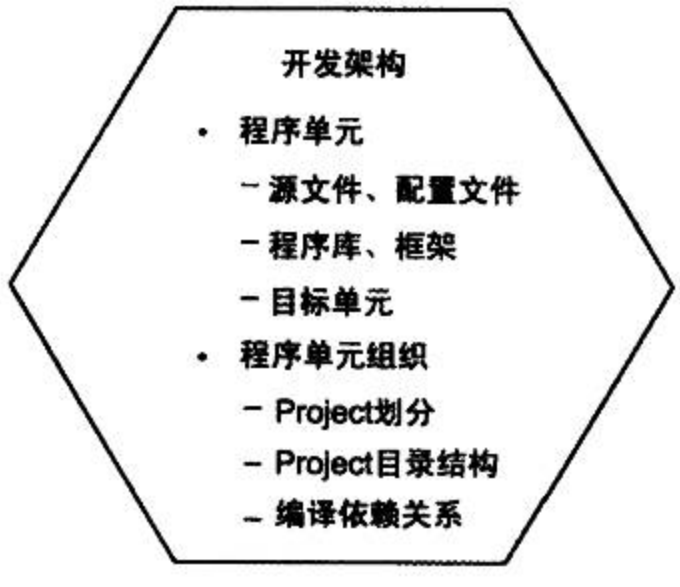


图 14-7　开发架构的设计内容

值得说明的是，此处的“Project”不是指 Project Manager 管的“项目”，而是指 IDE 等开发环境所支持的“Develop Studio Project”类似的概念（图 14-8 所示为 VC++支持的 Project）。

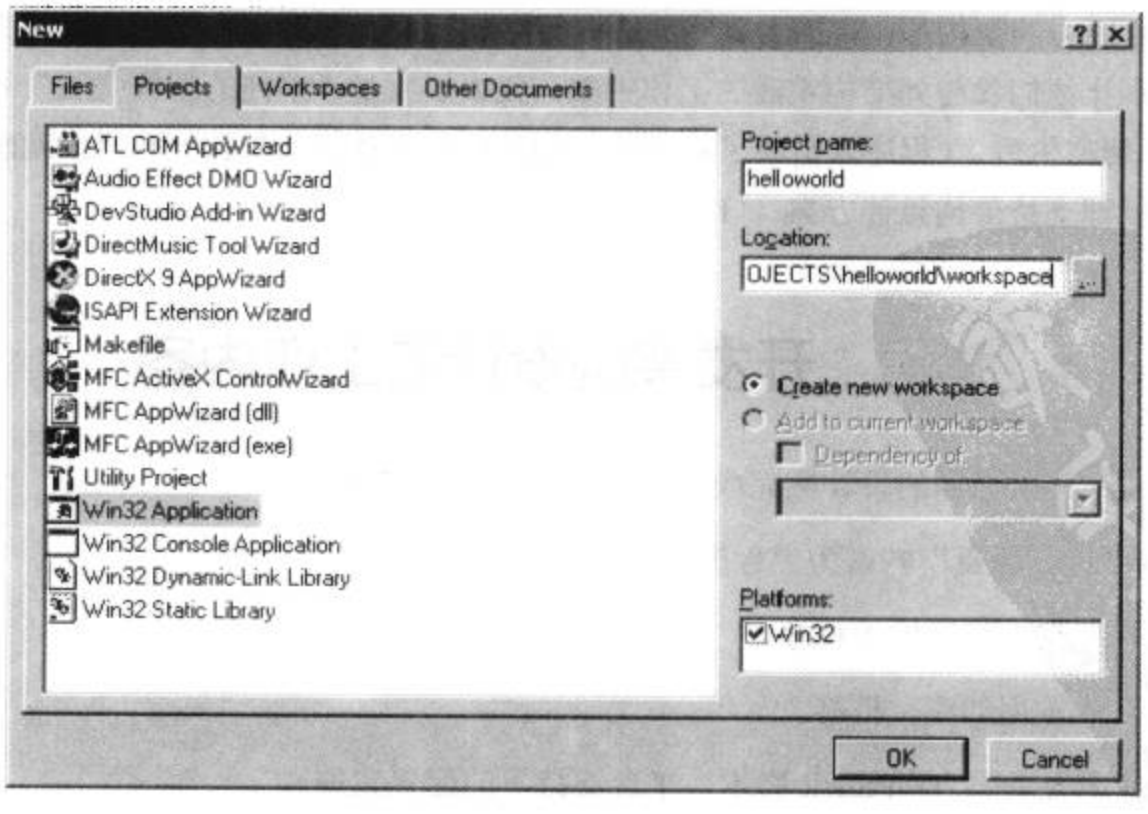


图 14-8　注意开发架构中“Project 划分”的含义

14.9 观点：重用测试是关键

广大软件企业希望通过为客户提供合格的软件系统达到获取利润的目标，于是，他们非常关注“可重用性”。但从重用的现实效果来看，显然远远不能令人满意……

14.9.1 探究：我们为何年复一年修改着类似的 Bug

在很多企业（甚至包括很多程序员本人）都声称“我们很重视重用”的背景之下，下面这个问题尤其值得深思：

> 国内许多程序员年复一年地写着类似的程序——更要命的是，他们年复一年地修改着类似的 Bug。

事出有因。下面两个问题是根源所在。

第一，架构师“重交付、轻维护”。心理学告诉我们，自己的、眼下的痛苦是最大的痛苦。于是，在工期的巨大压力下，许多架构师最担心的是“项目能否按时交付”，于是就把维护问题扔在脑后。

第二，架构师“重视小粒度重用、忽视大粒度重用”。一个孤立的类、一个小函数，每天都能被重用，却不能解决“年复一年修改类似 Bug”的问题。最终我们发现，小粒度重用是有价值的，但和大粒度重用并不矛盾，而大粒度重用才是避免“重复组装、重复改 Bug”式浪费的关键。所以，请记住这个公式：

> 重用价值 = 重用次数 × 单次价值

14.9.2 观点：为了从根本上降低维护成本，重用测试是关键

不要只关心重用的次数，为了重用而重用，而是时刻关注节省成本——这才是重用的目标所在。于是，当我们想到“维护是最昂贵的环节”，当我们看到经典数据显示维护成本占总成本的 67%之巨（如图 14-9 所示）时，会做何感想呢？

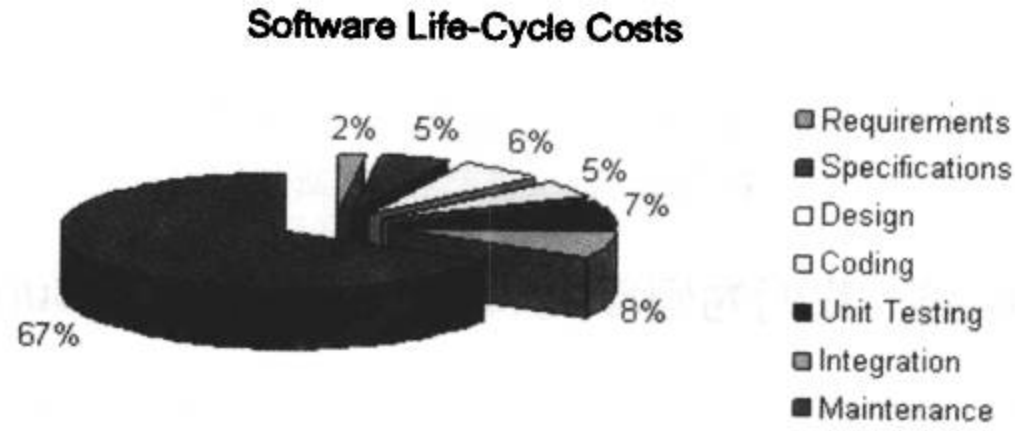


图 14-9　维护是最昂贵的，占总成本的 67%

想必你的结论和我一样：为了从根本上降低维护成本，重用测试是关键。这意味着大粒度重用。也就是说，在重用这些代码的时候，并不对代码进行任何修改——它们的测试也被重用了。具体而言，Framework、Service、Server、平台、中间件都算大粒度重用技术，它们已经成为，并继续是重用技术的未来方向。

14.9.3　简评：设计模式对重用的意义

最后，评论一个有趣的现象：很多架构师一提到重用首先会想到设计模式。那么，设计模式在重用技术中占据什么位置呢？先看看 Lethbridge 在《面向对象软件工程》中的一段话：

> 下面是软件工程师实践过的一些重用类型，按照重用所节省的潜在工作量的升序排列。
>
> - 重用专家经验。
> - 重用标准的设计与算法。
> - 重用类库或程序，或者重用语言和操作系统中内置的强大命令。
> - 重用框架。
> - 重用完整的应用程序。

设计模式属于上面的“专家经验”和“标准的设计”级别的重用策略——《面向对象软件工程》明白无误地告诉我们：这种“重用类型”节省的潜在工作量是比较有限的。

为什么呢？图 14-10 说明了“设计模式”和“框架”等技术在重用方面区别的根源：前者通用程度高和编程语言无关，后者实现程度高代码已提供。没有代码——无论是系统软件或平台内部的实现，还是库或框架这种外部实现——就没有办法重用测试，就会因面临较多 Bug 而花费较高的维护成本。

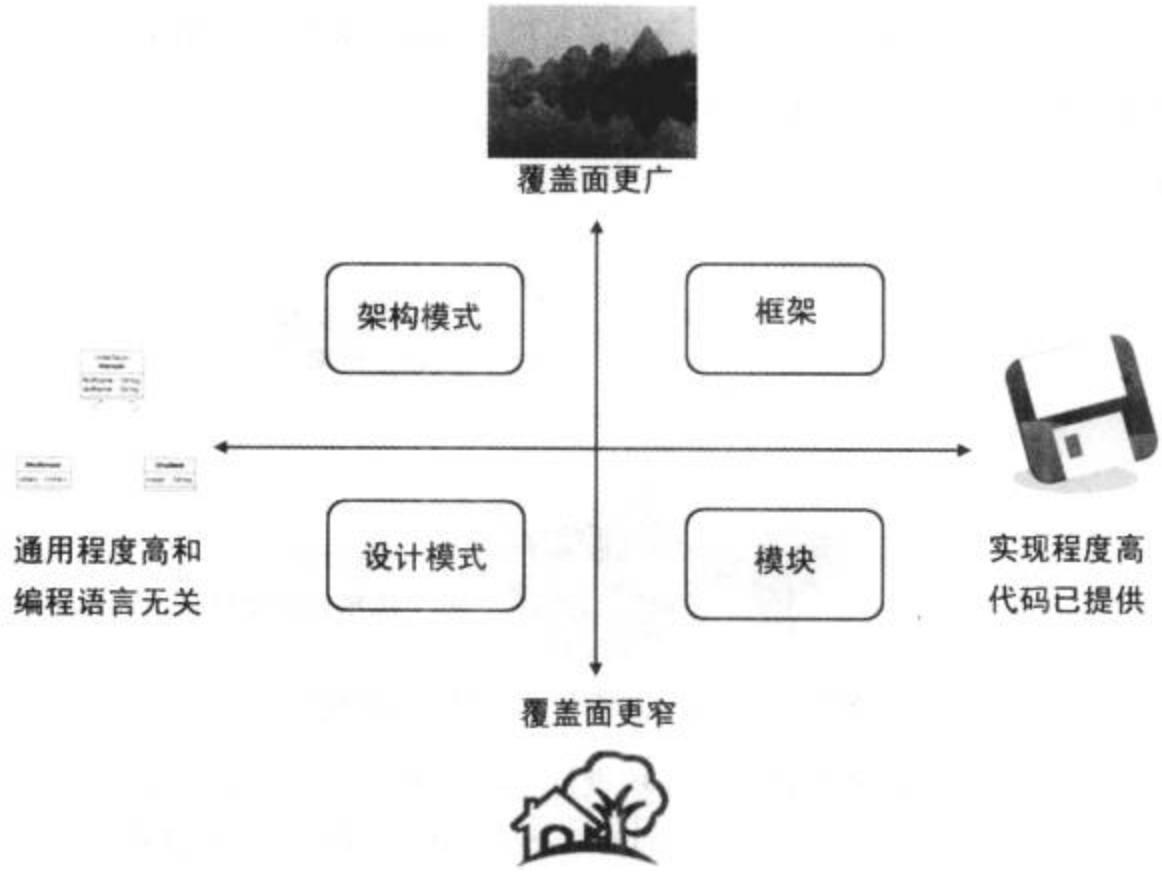


图 14-10 模块、框架、设计模式、架构模式的对比关系

14.10 贯串案例

继续 PASS 系统的架构设计。

14.10.1 物理架构

回顾第 9 章的“贯穿案例”部分，读者可能会问：物理架构不是在概念架构时已设计了吗？为什么又要设计？

所以，下面首先分析一下：概念架构设计和 5 视图方法是什么关系。

- 总体而言，概念架构所包含的高层设计决策终究不会跳出 5 个架构视图的范围——逻辑架构、物理架构、开发架构、数据架构或运行架构。
- 只不过，概念架构设计的抽象程度比较高，设计程度（图中的阴影代表工作量）也很不充分，而细化架构必须设计到可以指导开发的程度。
- 例如，一个分布式系统的概念架构，最常见的做法是包含逻辑架构和物理架构 2 个视图的高层次考虑。
- 对于典型的嵌入式控制系统而言，概念架构设计时又经常关注运行架构中多条控制流的规划。

总之，概念架构与 5 视图方法的区别及联系可以用两句话概括（如图 14-11 所示）：

- 概念架构从少数视角、重点视角进行概念级设计；
- 细化架构从多个视角、全面视角进行充分设计。

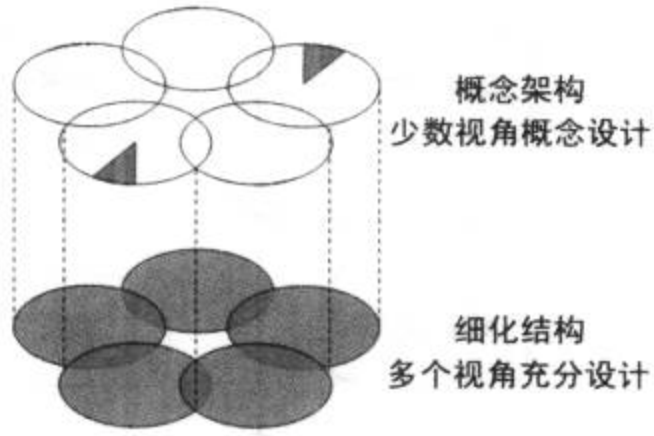


图 14-11　概念架构与 5 视图方法的关系

至此，我们明白了 PASS 系统的物理架构设计已经在概念架构时进行了一部分，进一步明确不同物理层（Tier）之间的协议类型、每一物理层是否须要部署群集、软件单元（例如 Layer）到物理节点的映射关系等物理架构设计的内容即可。图 14-12 显示了 PASS 系统 Layer 到 Tier 的映射关系。

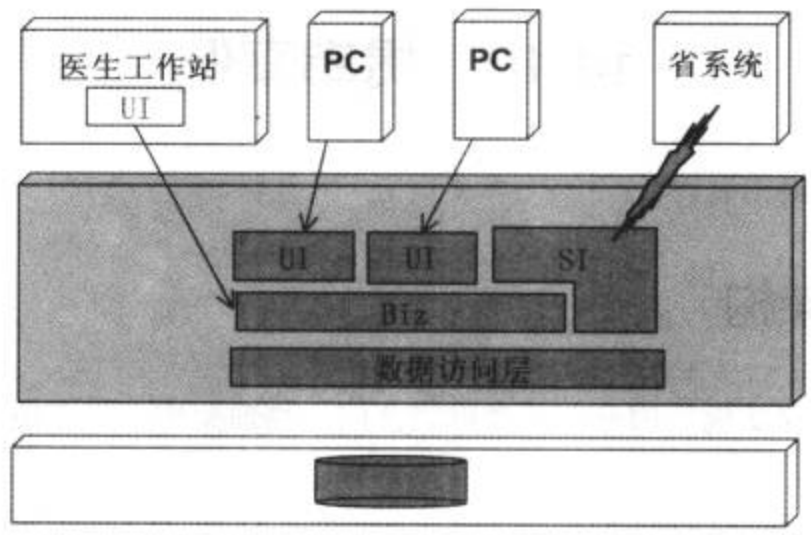


图 14-12　PASS 系统 Layer 到 Tier 的映射关系

14.10.2　持续不断地考虑非功能需求

非功能需求的支持不可能是“速决战”。ADMEMS 方法强调：

> 非功能需求的考虑是“贯穿环节”，在概念架构设计阶段，以及细化架构设计阶段都应重视。

对细化架构设计阶段而言，这意味着非功能目标的考虑时刻伴随着 5 视图的设计。例如，为了支持“用药规则的实时更新”这一功能，会涉及不同视图的考虑，中间穿插着无数次“质疑-分析-决策”的“微过程”。表 14-1 所示的目标-场景-决策表“重放”了我们的思维过程。

表 14-1 “用药规则的实时更新”相关的非功能考虑过程

目标	场景	决策
可管理性	• 省中心，发起用药规则更新，维护成百上千家医院 PASS 系统的远程访问地址成为棘手问题	否定省中心发起更新的方式
	• 每个 PASS 系统，发起更新，只需要依赖同一个相对不变的省中心访问地址	采用 PASS 系统发起更新的方式
数据一致性	• 规则更新到数据库的过程，可能出错，造成规则缺失或不一致	系统间的同步机制
性能	• （环境：已引入同步机制）规则更新，采用基于每单笔规则的同步机制，复杂且性能低	采用基于“每日规则集”的批量同步机制

14.10.3 开发架构

对大系统而言，开发架构设计中的“Project 划分”是不可或缺的，因为即使一个 Project 可以“胜任”（例如没有多节点因素），我们也不推荐这样做。更何况，将一个系统组织成多个 Project 的形式进行开发，可以方便地单独控制每一个 Project 源码的保密性——这是很多软件企业都关心的一个问题。

如图 14-13 所示，我们将 PASS 系统划分为 5 个 Project，分别为：

- 院长 Web 应用；
- 管理员 Web 应用；
- 嵌入医生工作站的 DLL；
- 医生模块通用 SDK；
- PASS Server。

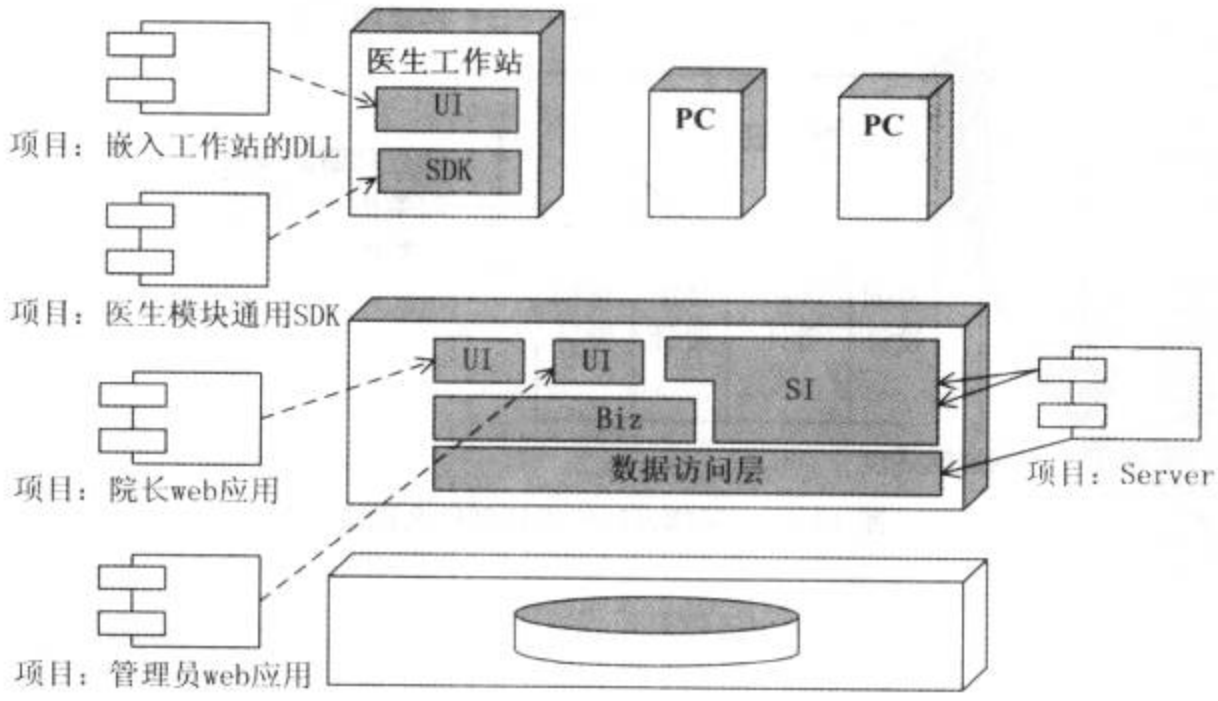


图 14-13 PASS 系统开发视图：Project 的划分

确定了 PASS 系统划分成哪几个 Project，就可以进一步确定每个 Project 所包含的程序单元、这些程序单元的依赖关系、源码的基本组织结构等问题，并为每个 Project 选定具体的技术及所用到的 Framework。

14.10.4　架构设计应进行到什么程度

最终，架构设计应进行到什么程度呢？

如图 14-15 所示，架构设计的程度应考虑 3 方面的问题。

- 应为开发人员提供足够的指导和限制。
  - 标志：可支持并行的详细设计。
  - 说明：所谓架构设计、概要设计、总体设计，只是含义相同的不同词汇而已。因此，“架构设计→概要设计→详细设计”的观点是不够专业的。
  - 解图：图中“架构设计”和“详细设计”之间没有缝隙。
- 因项目、开发团队情况的不同而变化。
  - 说明：应考虑项目熟悉程度、风险高低，以及团队技能水平等。
  - 解图：图中“架构设计”和“详细设计”之间的“线”上下浮动。
- 业务层、通用机制应更深入地设计。
  - 说明：核心模型影响可扩展性，应当更深入设计；通用机制影响易理解性和 Bug 率，应当更深入设计。
  - 解图：图中“架构设计”和“详细设计”之间的“线”为深深浅浅的锯齿状。

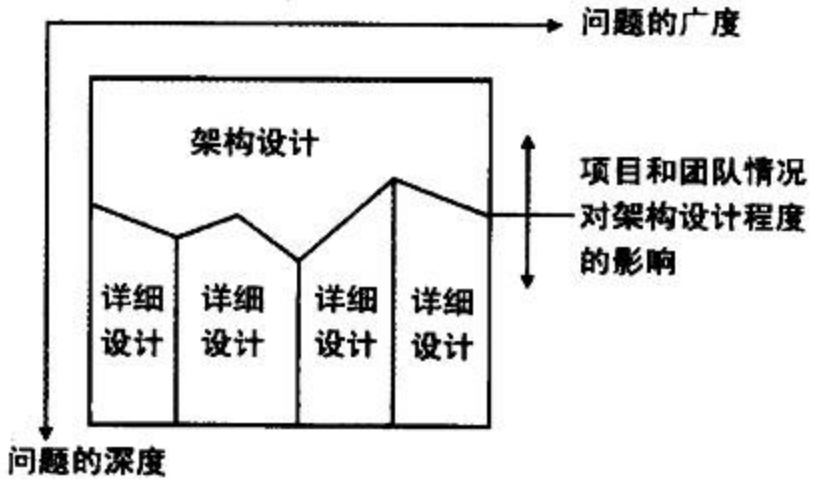


图 14-14　架构设计应进行到什么程度

第 15 章
数据架构的难点：数据分布

压力、没时间进行充分测试、含糊不清的规格说明……无论多努力工作，还是会有错误。不过，造成无法挽回失败的，是数据库设计错误和架构选择错误。

——Stéphane Faroult，《SQL 语言艺术》

针对不同的领域，由于信息资源类型及其存在的状态不同，信息资源整合的需求存在较大差异。

——彭洁，《信息资源整合技术》

确定数据分布方案是数据架构设计的难点。越是大系统，数据分布越关键。因此，一线架构师迫切须要建立数据分布策略的大局观。

本章结合案例，讲解如何运用数据分布的 6 种具体策略。

15.1 数据分布的 6 种策略

所谓分布式系统，不单单是程序的分布，还涉及数据的分布。而且，处理数据分布问题常常更加棘手。

根据系统数据产生、使用、管理等方面的不同特点，常采用不同的数据分布式存储与处理手段。总体而言，可以归纳为以下 6 种策略：

- 独立 Schema（Separate-schema）。
- 集中（Centralized）。
- 分区（Partitioned）。

- 复制（Replicated）。
- 子集（Subset）。
- 重组（Reorganized）。

15.1.1　独立 Schema（Separate-schema）

当一个大系统由相关的多个小系统组成，且不同小系统具有互不相同的数据库 Schema 定义，这种情况称为“独立 Schema”。

图 15-1 所示的示意图，说明了独立 Schema 方式的理解要点——“Application 不同，Schema 不同”。

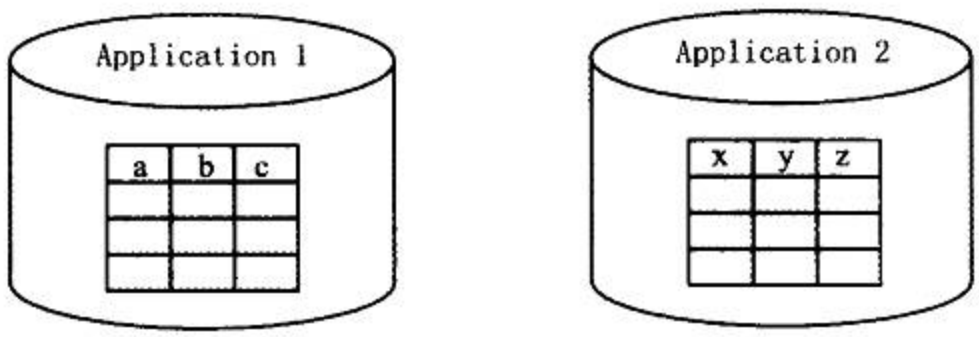


图 15-1　数据分布策略“独立 Schema”示意图

如果可以，架构师应首选此种数据分布策略，以减少系统之间无谓的相互影响，避免人为地将问题复杂化。

15.1.2　集中（Centralized）

指一个大系统必须支持来自不同地点的访问，或者该系统由相关的多个小系统组成，而持久集中化数据进行集中化的、统一格式的存储。

如图 15-2 所示，该方式的特点可用“集中存储、分布访问”来概括。

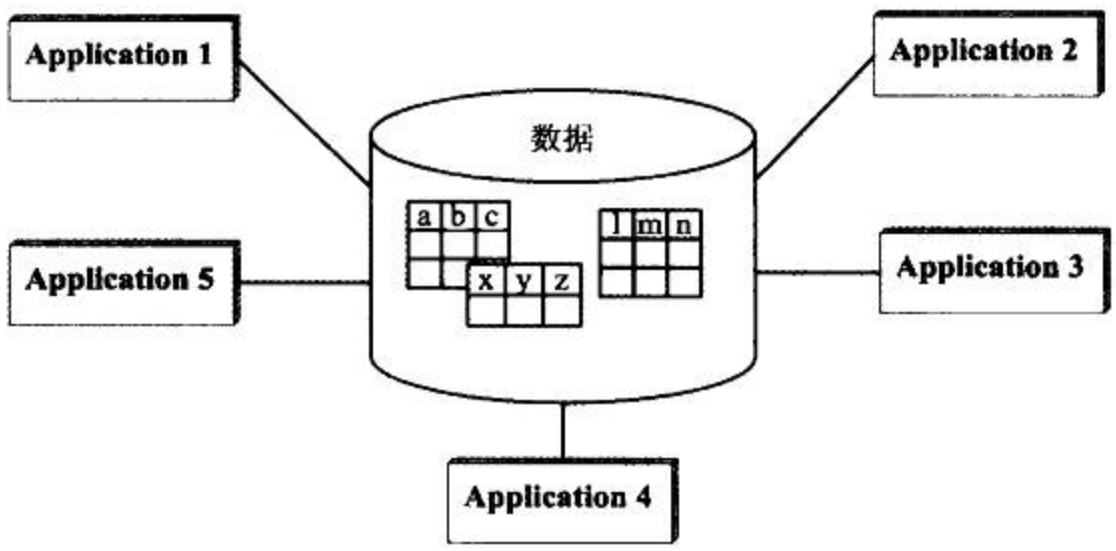


图 15-2　数据分布策略“集中”示意图

15.1.3 分区（Partitioned）

分区方式包含水平分区和垂直分区两种类型。

当系统要为“地域分布广泛的用户”提供“相同的服务”时，常常采用水平分区策略。如图 15-3 所示，水平分区的特点可以概括为“两个相同，两个不同”——相同的应用程序、不同的应用程序部署实例（instance），相同的数据模型、不同的数据值。

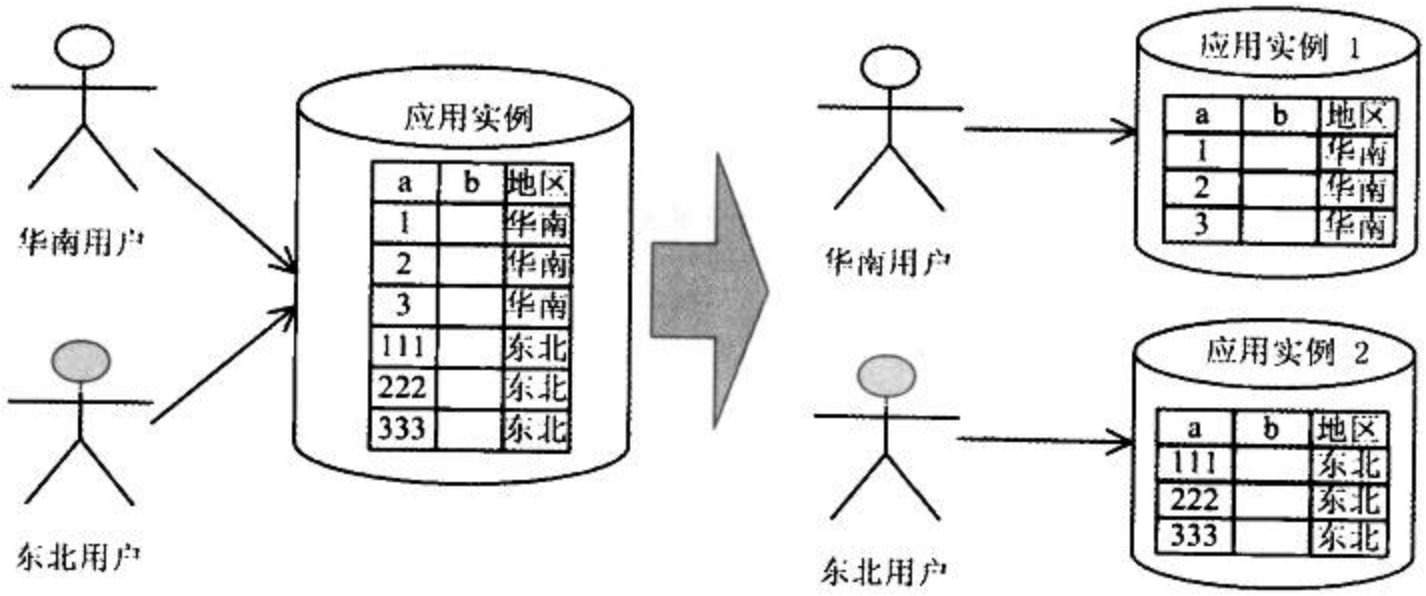


图 15-3 “水平分区”示意图

在实践中，水平分区的应用非常广泛，而垂直分区的作用相比之下要小得多。图 15-4 说明了垂直分区方式的特点：不同数据节点的 Schema 会有“部分字段（Field）”的差异，但可以从同一套总的数据 Schema 中抽取得到。

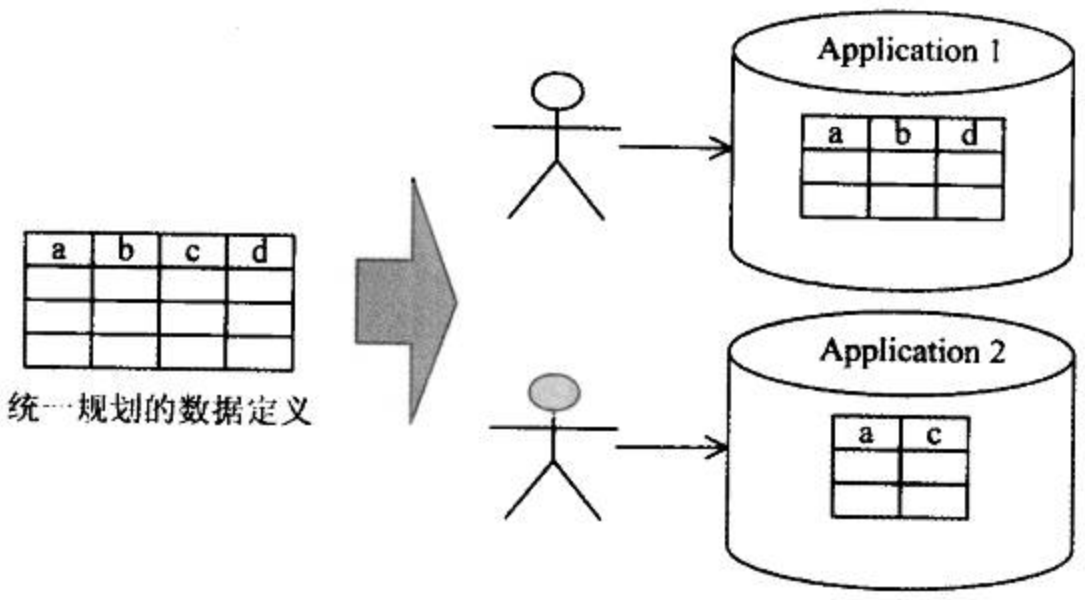


图 15-4 “垂直分区”示意图

另外，须要特别说明的是，本节所讲的“分区”不是指 DBMS 支持的“分区”内部机制——后者作为一种透明的内部机制，编程开发人员感觉不到它的存在，常由 DBA 引入；而本节所讲“分区”会影响到编程开发人员，或者应用部署工程师，一般由架构师引入。

15.1.4　复制（Replicated）

在整个分布式系统中，数据保存多个副本，并且以某种机制（实时或快照）保持多个数据副本之间的数据一致性，这就是复制方式的数据分布策略。如图 15-5 所示。

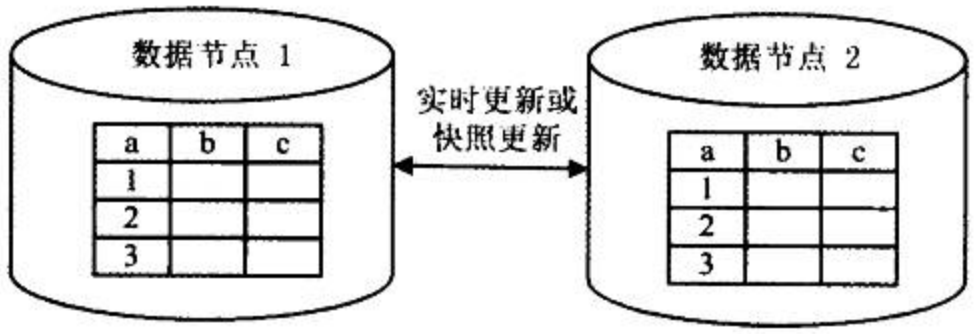


图 15-5　数据分布策略"复制"示意图

15.1.5　子集（Subset）

"子集"是"复制"的特殊方式，就是某节点因功能或非功能考虑而保存全体数据的一个相对固定的子集，如图 15-6 所示。

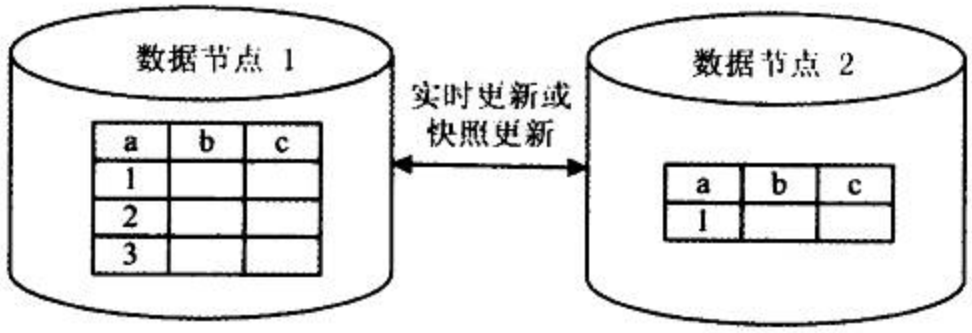


图 15-6　数据分布策略"子集"示意图

总体而言，子集方式和复制方式有着非常类似的优点：

- 通过数据"本地化"，提升了数据访问性能。
- 数据的专门副本，利于有针对性地进行优化（例如：支持大量写操作的 DB 副本应减少 index，而以读为主的 DB 副本可设更多 index）。
- 数据的专门副本，便于提高可管理性，加强安全控制。

不过，实践中常优先考虑子集方式，因为它与复制方式相比有两大优势：

- 减少了跨机器进行数据传递的开销。
- 降低了数据冗余，节省了存储成本。

15.1.6　重组（Reorganized）

业务决定功能，功能决定模型。当遇到数据模型不同时，一般都能够从功能差异的角度找到

答案。

重组这种数据分布策略，就是不同数据节点因要支持的功能不同，而以不同的 Schema 保存数据——但本质上这些数据是同源的。于是，重组策略须要进行数据传递，但不是数据的“原样儿”复制，而是以“重新组织”的格式进行传递或保存，如图 15-7 所示。

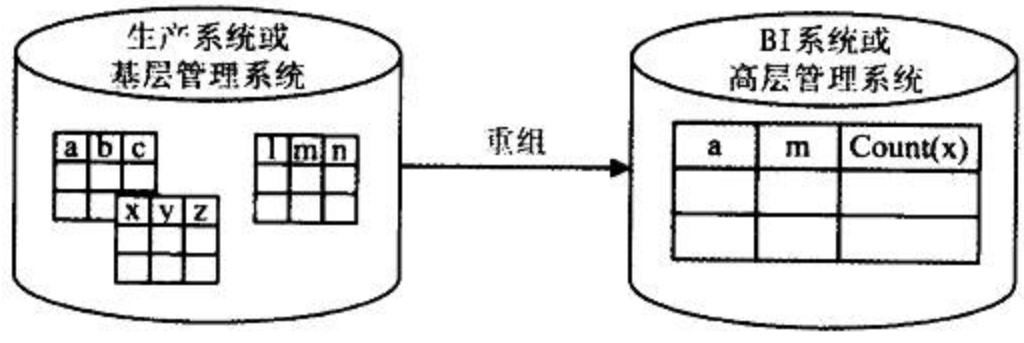


图 15-7 数据分布策略“重组”示意图

根据典型的应用背景，本书将重组分为两种类型：统计性重组、结构性重组。

例如，如果总公司只须要掌握各分公司的财务、生产等概况信息，那么就不须要把下面的数据原样复制到总公司节点，而是通过分公司应用对信息进行统计后上报。这叫“统计性重组”——数据的重新组织较多地借助了抽取、统计等操作，并形成新的数据格式。

“结构性重组”的例子，最典型的就是 BI 系统。生产系统的数据被进行整体重组，增加各种利于查询的维度信息，并以新的数据 Schema 保存供 BI 应用使用。

15.2 数据分布策略大局观

没有大局观，就很难理性选择数据分布的策略。因此，我们来总体对比 6 种数据分布策略的相同点及不同点。

15.2.1 6 种策略的二维比较图

一图胜千言，再次借助图来揭示“复杂背后的简单”，如图 15-8 所示。

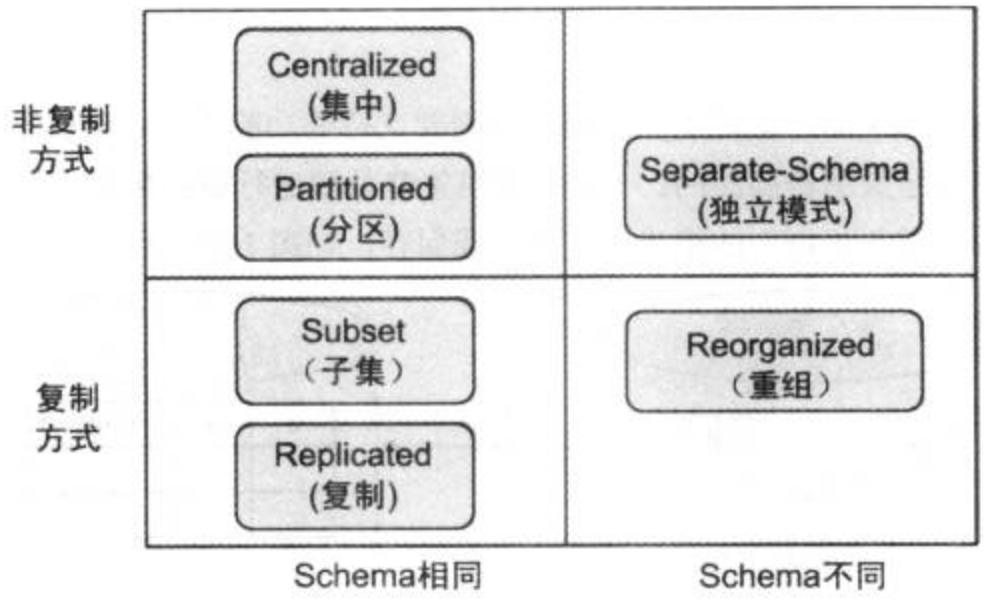


图 15-8　6 种策略的二维比较图

如图 15-8 所示，根据系统的特点不同，架构师所规划的数据分布策略无非分为两种方式：非复制方式、复制方式。

非复制方式包括 3 种具体策略：集中、分区、独立 Schema。

复制方式也包括 3 种具体策略：复制、子集、重组。

图中的另一个对比视角是：数据节点的 Schema 是否相同。其中，独立模式和重组两种方式，像它们的名字暗示的那样采用了不同的数据 Schema，而集中、分区、复制、子集这 4 种方式统一使用了相同的数据 Schema。

15.2.2　质量属性方面的效果对比

选择数据分布策略，应特别关注它们在质量属性方面的效果。

下面进行“冠军评选”，看看哪些策略分别在可靠性、可伸缩性、通信开销、可管理性，以及数据一致性等方面表现最佳（如表 15-1 所示）。

表 15-1　可靠性、可伸缩性、可管理性比较示意图

	可靠性	可伸缩性	通信开销	可管理性	数据一致性
独立 Schema			√	√	
集中				√	√
分区（指水平分区）		√			
复制	√				
子集					
重组					

可靠性冠军：复制。冗余不利于修改，但有利于可靠性。总体而言，复制方式的数据分布策略是可靠性的冠军。

其实，复制方式的可靠性和最终的“复制机制”密切相关，例如每天以快照方式来同步数据，

不如实时同步的可靠性高。

可伸缩性冠军：（水平）分区。Scale Up 会随着服务规模的增大变得越来越昂贵，而且它是有上限的。对超大规模的系统而言，Scale Out 是必由之路。而（水平）分区的数据分布策略非常方便支持 Scale Out。

有的文献上说“复制”方式对可伸缩性的支持也非常高，这种观点只对了一半——当数据以只读式“消费”为主时，通过复制增加服务能力的效果才好，否则为保证数据一致性而进行的“写复制”会消耗不少资源。

通信开销冠军：独立 Schema。独立 Schema“得这个奖”是实至名归。这很容易理解，既然独立 Schema 方式强调“将一组数据与它关系密切的功能放在一起”的高聚合原则，那么覆盖不同功能范围的应用之间就是松耦合的——用于传递数据的通信开销自然就小了。

可管理性冠军：独立 Schema。是的，还是它！由专门的数据 Schema 分别支持不同的应用功能，它们是相对独立的，便于进行备份、调整、优化等管理活动。

因此，本章前面提到：“如果可以，架构师应首选此种数据分布策略，以减少系统之间无谓的相互影响，避免人为地将问题复杂化。”

可管理性冠军（并列）：集中。为什么会存在“并列冠军”呢？因为从绝对角度评价可管理性是没有实际意义的。对采用了“数据大集中”的超大型系统而言，数据中心的管理工作依然颇具挑战性，但相对于分散的存储方式而言可管理性已大有改观。可管理性应该视原始问题的复杂程度而论，是相对的，而不是绝对的。

数据一致性冠军：集中。所有用户面对同样的数据，免去了修改同一数据不同实例的“麻烦”，便于保证数据的一致性。

15.3 数据分布策略的 3 条应用原则

至此，我们已全面了解了数据分布的 6 种策略。下面，借助案例介绍数据分布 6 大策略的 3 条应用原则：

- 把握系统特点，确定分布策略（合适原则）。
- 不同分布策略，可以综合运用（综合原则）。
- 从“对吗”、“好吗”两方面进行评估优化（优化原则）。

15.3.1 合适原则：电子病历 vs. 身份验证案例

合适的才是最好的。“把握系统特点，确定分布策略”，这是再明白无误的基本原则了。

医疗信息化中的电子病历可以复制，而各种系统常涉及的身份管理信息最忌讳复制。但为什么呢？

一句话，这是由系统的特点决定的。病历常作为医生诊断和治疗疾病的依据，是很有价值的资料。通过电子病历，可以将医务人员对病人患病经过和治疗情况所作的文字记录数字化，因此，电子病历的基本内容属于只读数据。为解决下列问题，电子病历可采用复制策略（例如在全省设置 3 个电子病历数据中心）：

- 各医院地域分布广，容易受到各种网络传输问题的干扰。
- 如果不能使用专网，还要考虑“跨网络”性能差的问题。

相反，身份管理信息不适合采用复制方式。用户信息有很强的修改特性：

- 新用户注册，意味着将有数据 Insert 操作。
- 用户修改密码或其他信息时，将有 Update 操作……

这时，如果采用复制，会造成大量数据同步操作。

所以，身份管理信息要集中，电子病历可以通过复制来提升性能和可访问性。如图 15-9 所示。

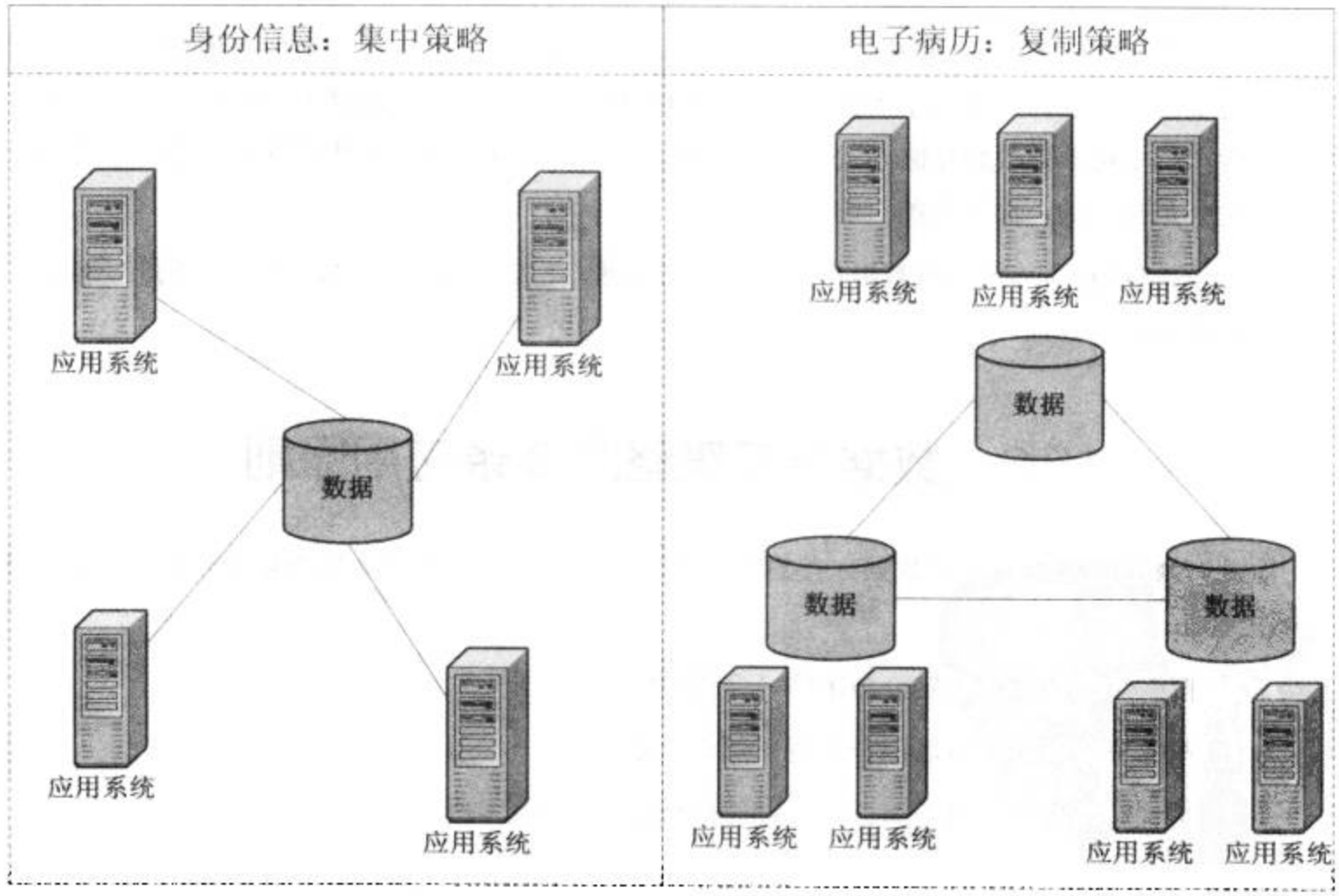


图 15-9　把握系统特点，确定分布策略（合适原则）

15.3.2 综合原则：服务受理系统 vs. 外线施工管理系统案例

当系统比较复杂时，其数据产生、使用、管理等方面可能很难表现出“压倒性”的特点。此时，就须要考虑综合运用不同数据分布策略。

电信 BOSS（业务运营支撑系统）是电信运营商的一体化支持系统，它主要由网络管理、系统管理、计费、营业、账务和客户服务等部分组成。信息资源共享是 BOSS 系统规划时的核心问题之一。

图 15-10 所示，为客户申请服务开通所涉及的业务流程。

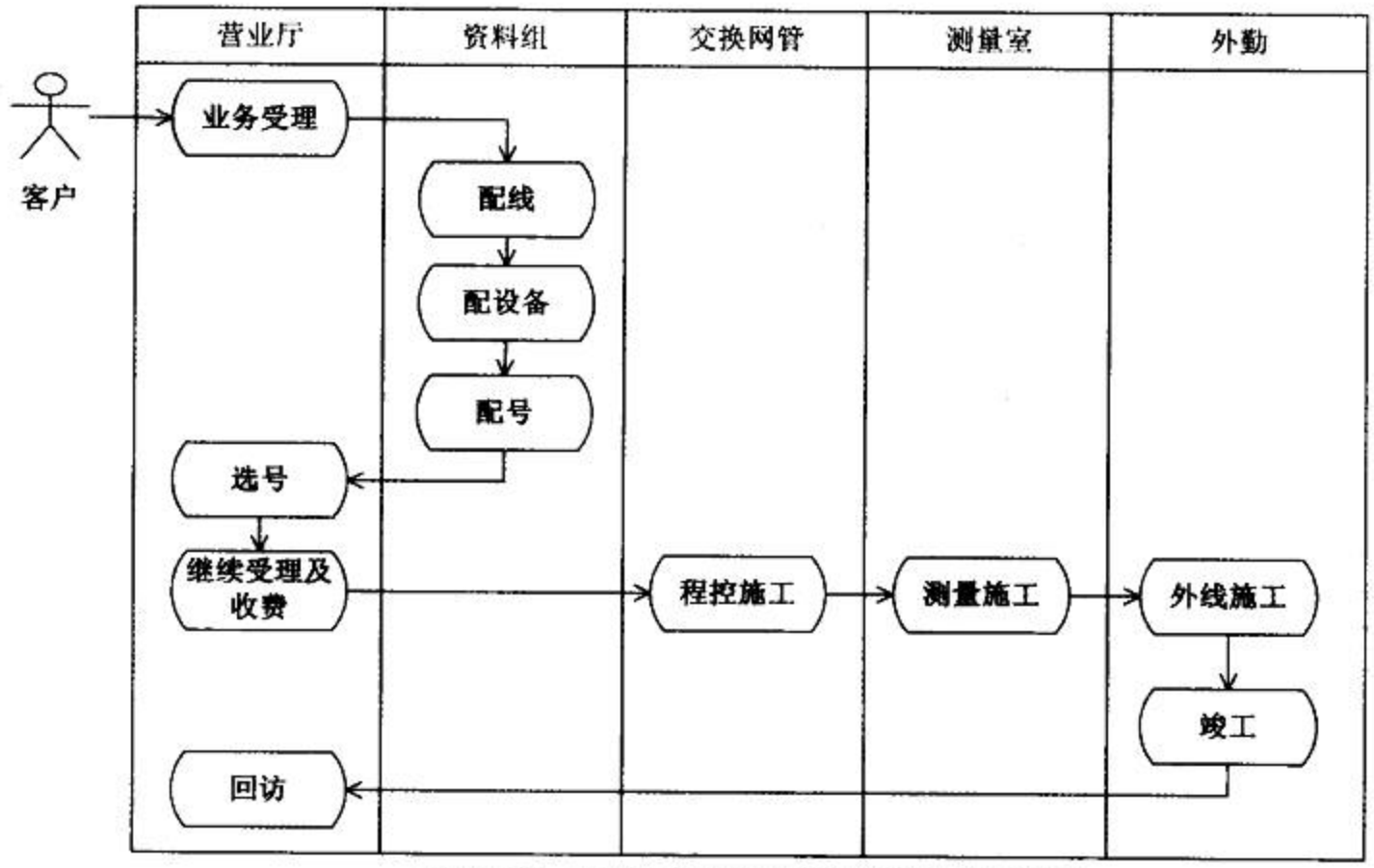


图 15-10 客户申请服务开通所涉及的业务流程

其中自己“服务受理系统”和“外线施工管理系统”两个系统所覆盖的业务是相对独立的，各有自己的业务数据，采用“独立 Schema”这种数据分布策略非常合适。至于两个系统之间存在互操作的关系，通过远程服务调用等形式支持即可。

单就“S 市电信服务受理系统”而言，应采用什么数据分布策略呢？考虑系统欲达到以下目标：

- 服务受理系统，应提供跨全市各辖区的、统一的服务。这意味着，在全市任何一家营业厅，都应该可以受理任何一个小区的电话开通业务。
- 例如，一个客户在浦东区居住，但在杨浦区上班，服务受理系统必须支持该客户在杨浦区申请开通浦东区某小区的一部固定电话。
- ……

所以，数据应集中。

再考虑“外线施工管理系统”。从业务角度，外线工作是典型的“划片分管”模式，一般由支局负责。所以，推荐外线施工管理系统“开发一套，多点部署”——数据分布策略是：水平分区。

总结一下，本例综合应用了 3 种数据分布策略（如图 15-11 所示）：

- 独立 Schema——服务受理系统和外线施工管理系统的数据相互独立。
- 数据集中——服务受理系统。
- 水平分区——外线施工管理系统。

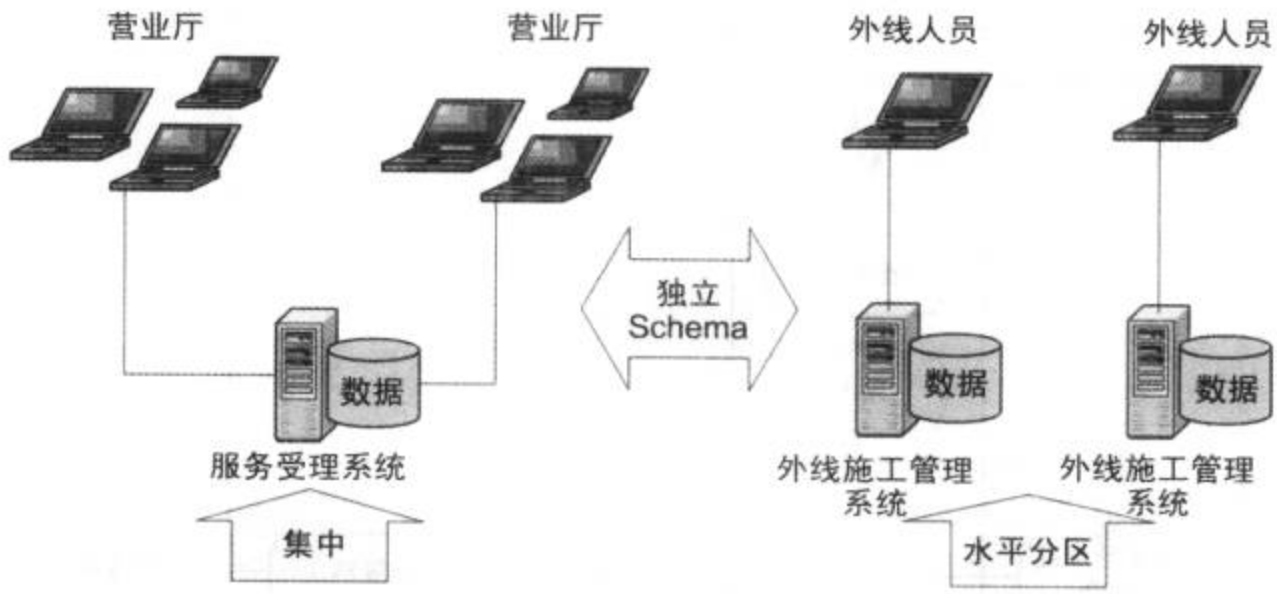


图 15-11　不同分布策略，可以综合运用（综合原则）

15.3.3 优化原则：铃声下载门户案例

> 架构设计是一个过程，合理的架构往往需要团队甚至外部的意见，因此注重优化原则很重要。一个有用的技巧是：当难以“一步到位”地做出数据分布策略的正确选择，以及还存在质疑时，应从“对吗”、“好吗”两方面进行对比、评估、优化。

关于铃声下载门户的数据存储方案，张、王、李、赵 4 人分别提出了 4 种方案：分区、复制、子集、集中。他们各不相让，几乎要吵起来了……

解决分歧、优化设计的办法是从“对吗”、“好吗”两方面进行对比评估。思维过程如表 15-2 所示：

- 分区。在功能支持方面，没有任何问题。但是在非功能方面不好，例如没有解决性能的问题。
- 复制。在功能支持方面依然没有任何问题。但是太贵，大量并不流行，甚至无人感兴趣的铃声被多次复制是毫无意义的。

- 子集。在非功能方面有着独有的优势，将部分流行的铃声在多点进行复制存储既促进了性能，又没有增加过多成本。但子集方式必然是一种辅助方式，因为它需要和另一种支持所有铃声保存的策略一起使用。
- 集中。用它和子集策略“搭配”，最为合适。

表 15-2 从“对吗”、“好吗”两方面进行对比评估，利于找到优化的方案

	提出者	对吗？（功能方面）	好吗？（非功能方面）	备 注
分区	小张	100	30	略显盲目
复制	老王	100	20	优点：性能高。缺点：浪费惊人
子集	小李	30	90	二八原则，热门铃声性能高。
集中	小赵	100	50	比铃声分区式分布存储简单

如此一来，总体的数据分布策略方案呈现出来：集中策略 + 子集策略。图 15-12 所示的架构图说明了这一点。

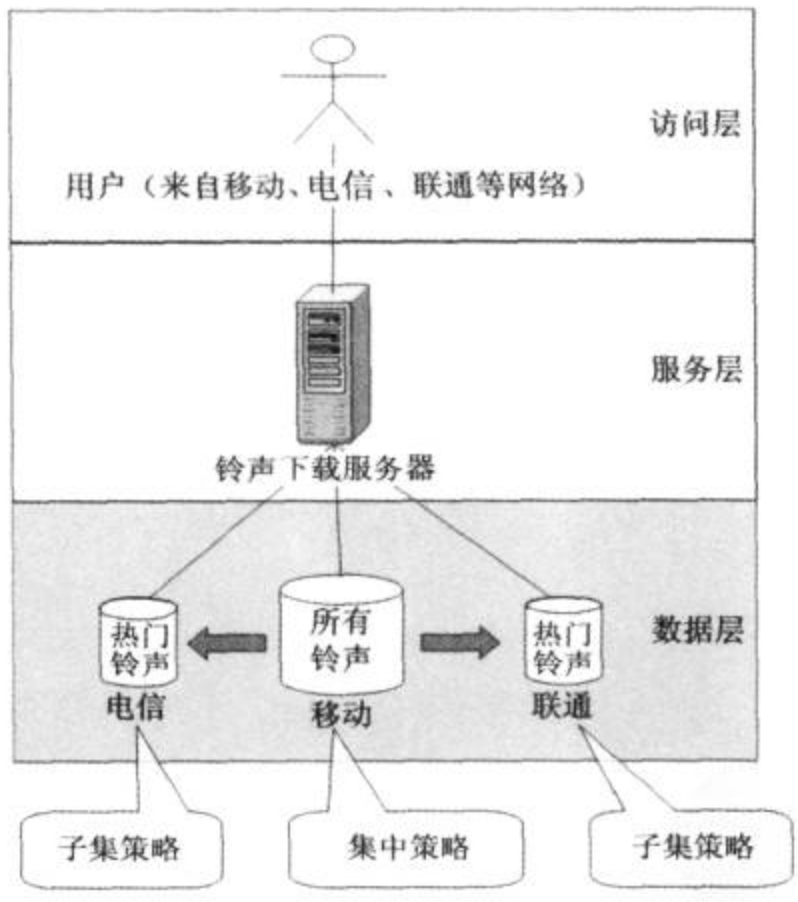
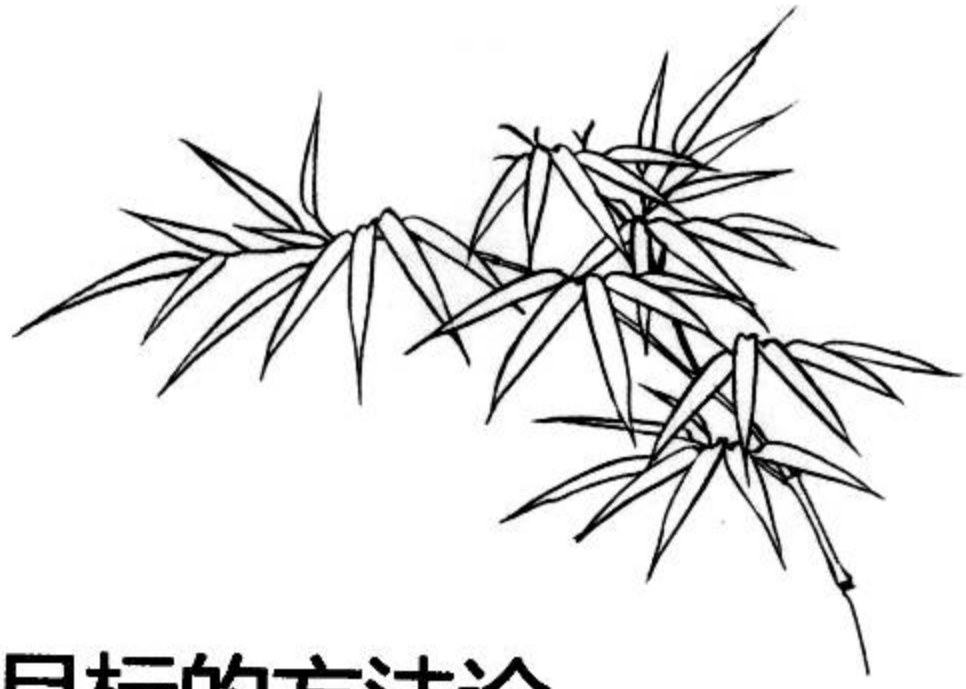


图 15-12 铃声下载门户的数据分布策略



专题：非功能目标的方法论

第 16 章
故事：困扰已久的非功能问题

非功能需求的满足程度对软件项目的成功非常关键……非功能需求分为质量属性和约束两大类。质量属性是软件系统的整体质量品质——所谓整体品质，就是它往往和大多数功能都有关，而不是仅仅表现在某个功能“内部”。至于约束性需求，它们是架构设计中必须遵循的限制、并有可能“衍生”出质量属性需求和功能需求。

——温昱，《软件架构设计》

那么非功能需求方面常见的问题是什么呢？……很多《需求规格说明书》中，会通过一个名为“设计原则”的小节来说明非功能需求，列出诸如高可靠性、高可用性、高扩展性等要求。但是很多开发人员根本就不去看它，因为这样的定性描述是没有判断标准的，因此这种信息传递是无效的。

——徐锋，《软件需求最佳实践》

软件架构设计为什么这么难？因为架构设计不是简单地处理“纯粹的技术问题”，而是要面对“技术与业务的关系问题”。最终，要求架构师不仅懂技术、懂业务，而且能理顺复杂的技术和业务之间的关系。

从面向业务的需求，到最终的面向技术的软件系统，要跨越很大的鸿沟。软件架构设计就是要完成从面向业务到面向技术的转换，在鸿沟上架起一座桥梁。软件架构师根据各种需求进行架构设计，最终的软件架构包含了结构、协作、技术等方面的重要决策，为系统化的开发活动建立了基础（如图 16-1 所示）。

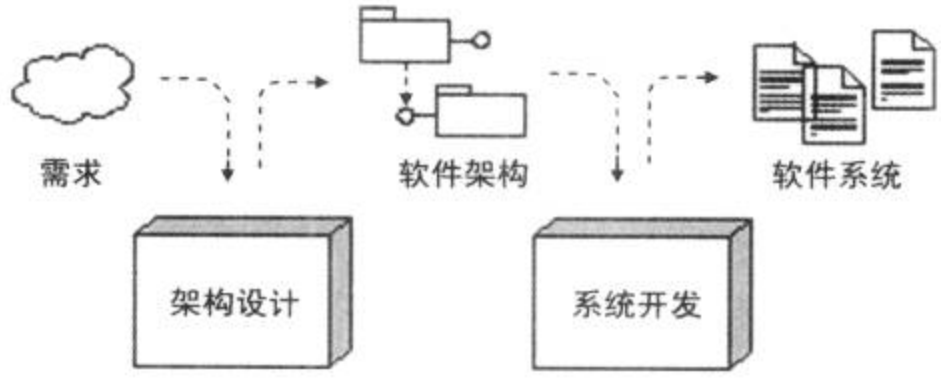


图 16-1　架构设计：跨越现实世界到计算机世界的桥梁

下面，请读者随着本章递进展开的 3 个故事，思考如下几个问题：

- 架构师是否应该懂需求呢？
- 功能需求、非功能需求都要懂吗？
- 生搬硬套需求标准是“懂需求”的应有表现吗？
- 应当如何有效传达非功能目标的具体要求？

16.1　“拜托，架构师不是需求分析师”

16.1.1　故事：小魏请教老沈

为数不少的架构师都认为自己是技术人员，职责是“知道需求之后设计出架构”。比如，老沈。

老沈是资深架构师，在某公司的设计部任职。一次，他手下的小魏充满虔诚地捧着几道题目来请教他（如图 16-2 所示）。

1.当《需求文档》中仅要求“高性能”时，架构师应注意进一步弄清用户的真实需求，因为“性能”含义宽泛，它可包括： A. 速度 B. 吞吐量 C. 持续高速性 D. 效率	2. 效率（Efficiency）和性能（Performance）的含义并不相同。例如大型机运行小软件，性能高、效率差。效率包括： A. CPU 使用率 B. 内存使用率 C. 硬盘 IO D. 网络 IO E. 速度

图 16-2　架构培训中的 2 道练习题（答案分别为 ABC、ABCD。）

老沈扫了一眼，皱起了眉头，语重心长地说：“小魏，你的理想不是要当首席架构师吗，怎么研究起需求来了？小伙子不要朝三暮四哟！”

小魏的表情充满了惊讶。稍过片刻，他鼓起十二分的勇气问道：“沈老师，您的意思是架构师不必深入研究需求？”

老沈道：“拜托，懂点儿软件工程好不好，现代软件工程讲究角色分工和团队合作，架构师不是需求分析师……”

16.1.2 探究：架构师必须懂需求

其实，除非特别简单的系统，否则架构师不能“吃透”需求，必然造成最终的系统无法很好地满足需求。但业界普遍存在的现实是：许多架构师的需求功底太差，输在了架构设计的“起跑线”上。当然，架构师也有一大堆“苦衷”：

架构师不能“吃透”需求，的确出人意料。但是既然大部分企业为架构师安排了“技术晋升路线”，既然许多架构师也把自己当“纯粹的技术人员”，既然架构师必须研究“时髦技术”否则就被程序员看不起，既然设计模式和 UML 还在“排队”，需求嘛就算了，那么架构师不能“吃透”需求，也就在情理之中了。

调侃归调侃（其实不无道理），但笔者一直认为：精通需求，是对架构师最基本的要求之一，不了解需求是现在许多架构师职业发展道路上的瓶颈。

值得强调的是，需求分析师和架构师这两个角色要掌握的需求知识并不完全一样（如图 16-3 中 B 所示）。经典的观点是认为架构师应掌握的需求知识是需求分析师的一个子集（如图 16-3 中 A 所示），其实不然：

- 例如，第 3 章所讲的“不同需求影响架构的不同原理”，需求分析师可以不去研究。
- 再例如，《软件工程的事实与谬误》中指出：“从需求转入设计时，因为制定方案过程的复杂性，会激增出大量的衍生需求（针对一种特定设计方案的需求）。设计需求是原始需求的 50 倍之多。”

A. 误解　　B. 正解

图 16-3 两个角色要掌握的需求知识并不完全一样

总之，架构师必须懂需求。虽然无须研究诸如需求捕获等技术，但需求类型、需求影响架构的原理、质量属性间的相互影响关系等都是必须精通的。

16.2　“敢说 ISO 9126 不对，真牛”

16.2.1　故事：小冯与小汪的争论

小冯和小汪争得不可开交。

小冯是项目经理，他说：“不要随意扩大需求的 Scope，更不要搞需求镀金，因为这些不仅意味着成本增加，还可能造成工期延误。”

“是的。可是……”小汪是架构师，他的话说了半截儿就被打断了。

小冯抢着说：“所以，既然客户仅要求‘高可靠性’，我们就不能把它换成‘持续可用性’，更不应该随意扩大需求的范围，把安全性、可管理性都加上。别忘了，成本超了、工期误了，可都是我这个项目经理扛着。”

“像这种直接影响企业正常运营的系统，客户要的肯定是‘持续可用性’，而不仅仅是‘可靠性’。” 空气中已经有点儿火药味儿了，但小汪哪肯退让，手指着培训教材上的一页继续坚持（如图 16-4 所示），“再请问，分布式的系统如果安全性差，可靠性怎么可能保证呢！”

> 有些文档常不区分可靠性（Reliability）和持续可用性（Availability）。但其实，二者的不同非常明显，下列情况影响可靠性的有____，影响持续可用性的有____。
>
> A. 硬盘坏，导致系统宕机 10 小时
>
> B. 停机维护，系统处于未工作状态 8 小时
>
> C. 黑客捣乱，部分用户 4 个小时都无法登录
>
> D. 系统宕机后，定位软件故障花了 72 小时

图 16-4　架构培训中的练习题（答案分别为 A、ABCD。）

“《ISO 9126》的一级质量属性里就没有‘持续可用性’，而是‘可靠性’。”小冯说。

“国际标准就不会错吗？”小汪豪气冲天。

“敢说 ISO 9126 不对，真牛……”

16.2.2　探究：死抱需求标准，还是务实应变

科幻故事总是轻松的，现实中的故事却或多或少地让人感到压力。

作为架构师，你是否认为：架构师重视需求 ＝ 熟悉领域知识和业务？

对，但不全面——因为还要研究质量属性需求。那么，你是否又认为：懂质量需求 ＝ 了解《ISO 9126》呢？

本书的观点：重视标准，但在一定程度上必然要对之进行调整、扩充以适应实践要求。例如《ISO 9126》将质量属性描述成“树”，但实际上应该是“网”（如图 16-5 所示），安全性影响可靠性就是一例。

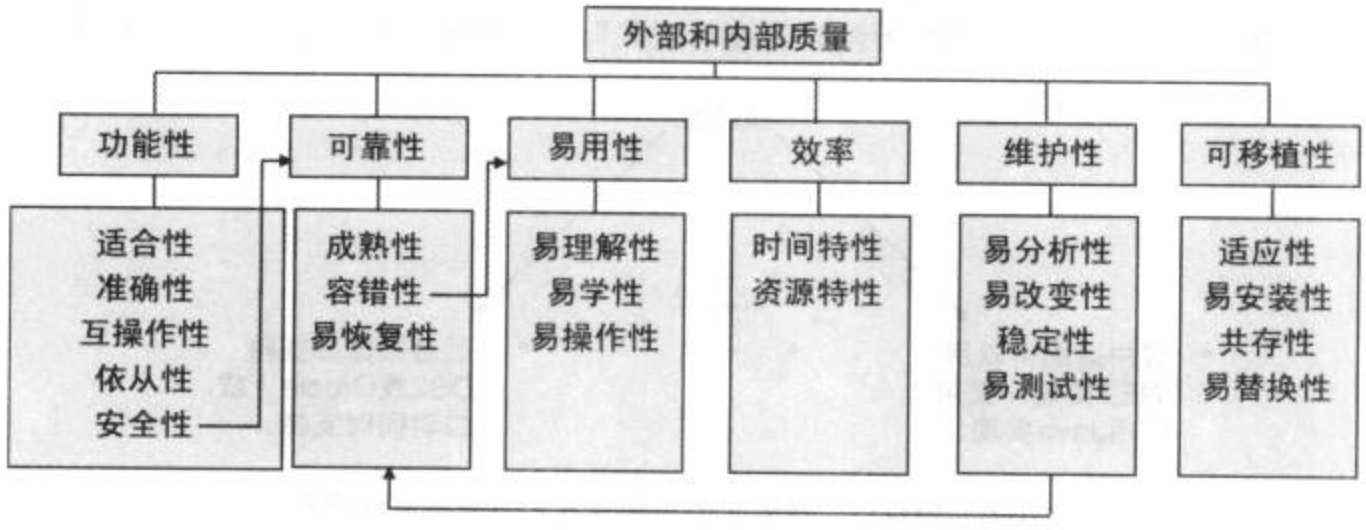


图 16-5 ISO 9126 将质量属性描述成“树”，但实际上应该是“网”

16.3 “我说得很清楚，架构要灵活”

16.3.1 故事：狮子说清了，绵羊没搞定

拿破仑说，“一只狮子率领一群绵羊的队伍，可以打败由一只绵羊带领一群狮子的部队”。听到这话，许多软件企业可能会觉得很高兴——因为不少公司都大量存在“一个狮子领导一群绵羊”这样的团队。

但就在本书完稿前不久，笔者和一客户沟通下一步培训时碰到的一个问题，就发生“狮子带领的绵羊团队”中。故事是这样的：

卢总资历深厚，水平挺高。此次公司立项研发一款新产品，卢总亲自担纲。架构设计前期，他对几个年轻的架构师一再强调，架构一定要设计得灵活。……产品很快进入了开发阶段，但随着几次需求变更的来临，卢总发现架构的灵活性较差。

最后，卢总对笔者说：“温老师你看，我的团队水平不高哇，我说得很清楚，架构要灵活，但最终还是过于僵硬。”

我说：“卢总，你谈的情况对定制培训很有帮助。同时我想说，将问题根源归于‘水平不高’其实只对了一半……”

16.3.2 探究：交流质量要求，如何做到“说得清楚、听得明白”

为什么说，卢总的分析只对了一半呢？因为卢总要求的“灵活性”这个目标过于笼统，既然

年轻的架构师水平偏弱，不能将架构灵活的要求“落地”也就不奇怪了。

那么，交流质量要求，如何做到“说得清楚、听得明白”呢？本书的建议是：场景化。例如，如图 16-6 所示，通过分析将“灵活性”更明确地诠释为“客户端即可以用.NET 实现又可以用 Java 实现”等一系列具体场景，对“狮子领导的绵羊团队”有效开展架构设计工作是大有裨益的。

图 16-6　场景化：将笼统的目标诠释为一系列具体场景

16.4　展望本部分的后续内容

3 个发生在我们身边的故事讲完了，呈现在我们面前的是层层递进的 3 个要求：

- 架构师必须懂需求（故事 1）。
- 需求 = 功能需求 + 非功能需求，架构师应同时关注两方面的需求，而且对质量的理解不应仅限于《ISO 9126》等标准（故事 2）。
- 业界的同行们在交流非功能需求的时候，普遍存在“说不透”的现象，架构师应有意识地克服（故事 3）。

在“专题：非功能目标的方法论”部分的后续几章，我们将着重介绍“场景”技术，以及“目标-场景-决策”表这种务实有效的、以满足非功能需求为目标的设计思维方法。

第 17 章
总论：非功能目标的设计环节

为了提高综合客户满意度及对不同质量属性的满意度，必须考虑计划和设计产品时的不同质量属性。

——Stephen H. Kan，《软件质量工程》

质量属性很难定义，但它们经常可以区分产品是只完成了其应该完成的任务呢，还是使客户感到很满意。……优秀的软件产品反映了这些竞争性质量属性的优化平衡。

——Karl E. Wiegers，《软件需求（第 2 版）》

作为决策者，架构师的工作影响着项目的成败，乃至公司的发展。我们须谨记：千万不要做“四拍”型的决策者。

决策时拍脑袋——就这么定了，

指挥时拍胸脯——保证没问题，

失误时拍大腿——我怎么没想到，

追查时拍屁股——老子不干了。

架构设计实践中，面对非功能需求目标时是最容易犯“拍脑袋”这个经典错误的。本章介绍非功能目标的设计思维及其实践要领。

17.1 非功能目标的设计环节简介

在我们当中，有不少人一厢情愿地认为：只要所开发出的系统完成了用户期待的功能，项目就算成功了，但这并不符合实际，忽视包括质量属性需求在内的非功能需求是很要命的。

为什么不少软件产品推出不久就要重新设计（美其名曰“架构重构”）？往往不是因为系统

“不能用”，而是由于系统架构“太拙劣”——从难以维护、运行速度太慢、稳定性差，甚至宕机频繁，到无法进行功能扩展、易遭受安全攻击等，不一而足。由此看来，软件的质量属性需求是不容忽视的，否则在大量的成本投入之后，很可能落得“失败”或“赔钱”的结果。

然而，软件的质量属性需求很“飘”，常常令架构师难以把握。如果缺乏足够的方法指导，即使勉强制定了设计决策也会觉得缺乏信心。

所以，解决问题的要害在于：如何使很“飘”的非功能目标被“明确定义下来”。如图 17-1 所示，分析如下：

- 需求决定架构，架构设计的成果已属于解决方案的范畴；
- 架构设计的过程是从“需求域”向“设计域”过渡的过程；
- 目标很飘，就意味着根据诸如“高性能”等非功能需求直接做出“设计决策”跨度过大了；
- 需要一种“过渡技术”来承上启下，它能使笼统的非功能目标明确化，它能帮助架构师做出更有针对性的设计决策；
- ——它，就是场景技术。

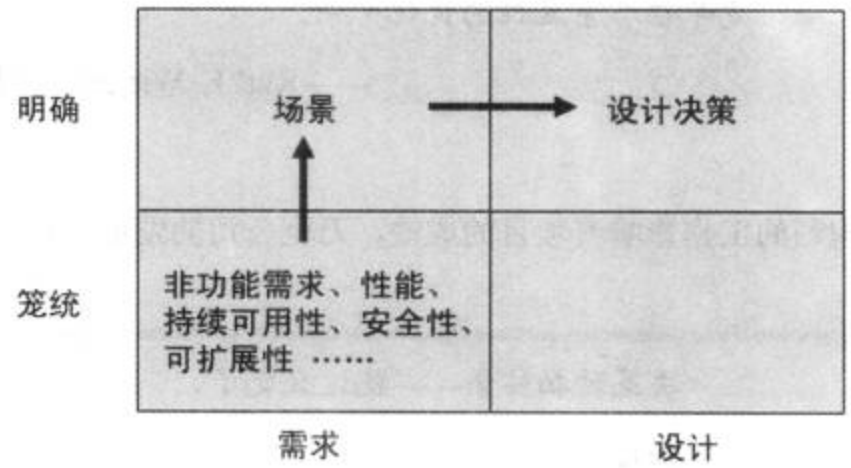


图 17-1　以场景为“跳板”的非功能目标设计思维

一句话，非功能目标的设计是以场景技术为核心手段、以目标-场景-决策表为思维工具，致力于支撑非功能目标的理性设计过程。

17.2　实际意义

作为架构师，有了非功能目标设计环节相关方法的指导，将获得如下几点优势：

- 设计更有针对性。

设计贵在务实，贵在有针对性。试想，如果世界上没有“高性能”这个高度简练、高度抽象的词，用户会怎么描述他的非功能要求？他一定会说一大堆“如果……则需要……”式的场景出来！

所以，将一个目标明确为 N 个场景，是一种回归本源的做法，可以使架构师的设计更加有针对性、更加有效。

- 可操作性强。

懂得了以场景为核心的非功能目标设计思维，架构师就知道“力往哪儿使”了。他们通过研究开发、维护、使用、变更等环节可能遇到的具体情况，不断发现场景，评估场景，做出决策……这是一个可操作性很强的思维实践过程。

其实，所谓方法，就是帮助你将经验更系统地、更充分地使用起来的思维框架。所以，对有经验的架构师而言，他们的思维更有序、经验的应用更有理有据、面对更大的系统时信心更足。

当前，很大程度上，确定支持非功能目标的设计决策的过程是“只可意会”的，例如阅读《架构设计文档》时很难搞清楚“为什么”，这给架构新手的成长造成了莫大的障碍。而现在，非功能目标的设计思维明确化了、可视化了、可操作化了，有利于架构新手学习、理解和掌握。

- 避免过度设计。

单凭经验为高质量属性而设计很容易造成过度设计，即引入的很多抽象和机制是不必要的，平白增加了设计复杂性。以“目标－场景－决策”表为工具的非功能目标设计方法将场景视为“一等公民”，使架构师很容易对非功能场景进行评估，通过权衡场景发生的几率、支持场景带来的价值、遗漏场景的代价等因素，来理性决定是否应支持该场景。

- 便于系统升级时参考。

当系统架构不能适应新要求时，往往要对架构进行重构，此时软件架构师常犯的毛病是过于强调系统架构的缺点，而将过去的架构设计全盘否定，这样可能造成设计出的新架构在解决了新问题的同时也失去了已有的优点。而如果将“目标—场景—决策”表文档化，则有利于避免上述问题。

17.3 业界现状

软件行业发展到今天，依然比较年轻。一个有趣的印证就是我们经常拿自己的行业和其他行业类比——今天类比建筑行业，明天类比汽车行业，后天类比拍电影，有趣！

我并不能确定把架构师和哪个职业相类比最合适，但或许，架构师最嫉妒的职业是拳击。人家的目标永远明确：打倒对方。而架构师，却要面对纠结在一起的“需求”——需求不是一个攻击目标，而是一堆攻击目标，而且是一堆可能不够明确、相互矛盾的目标。昨天、今天，甚至明天，都会有相同的故事在上演：笼统界定的“非功能目标”让软件架构师深感困惑……

这就是现状。

在这种现状之下，架构师不应“坐等”明确的需求，而是应该运用目标—场景—决策表等方法主动出击，设计成更有针对性的架构。

17.4　实践要领

17.4.1　场景思维

非功能需求支持是否到位，关键要靠场景思维的运用。图 17-2 归纳了场景思维在非功能目标设计中的重要性。

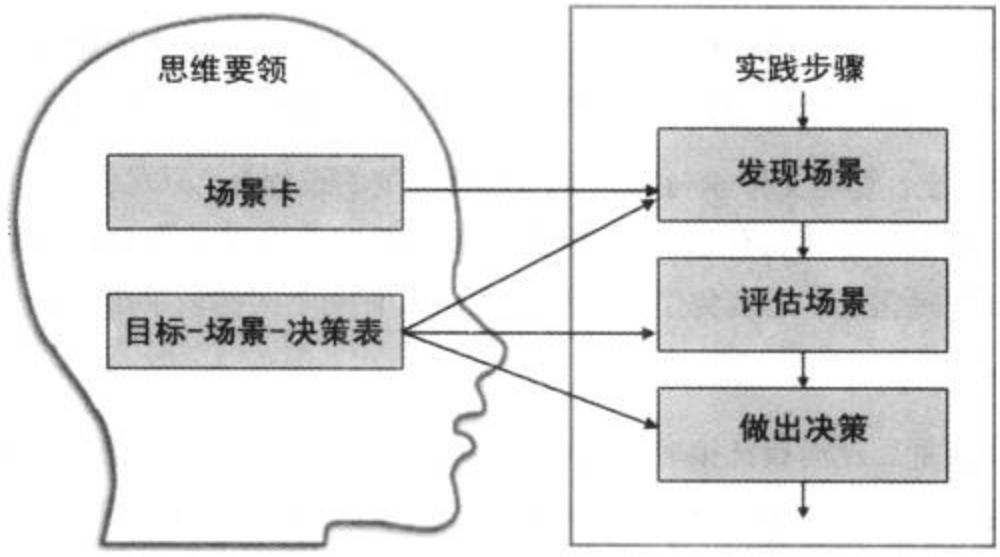


图 17-2　场景思维是方法核心

17.4.2　纵穿环节

有著名厂商认为，架构设计的“第几步”应该是非功能考虑，这种观点是危险的。

非功能需求不可能是“速决战”，它必然是“持久战”，连编码都会影响到性能等非功能属性，更何况概念架构设计和细化架构设计呢？所以架构师须注意，非功能目标的考虑是纵穿架构设计始终的环节，如图 17-3 所示。

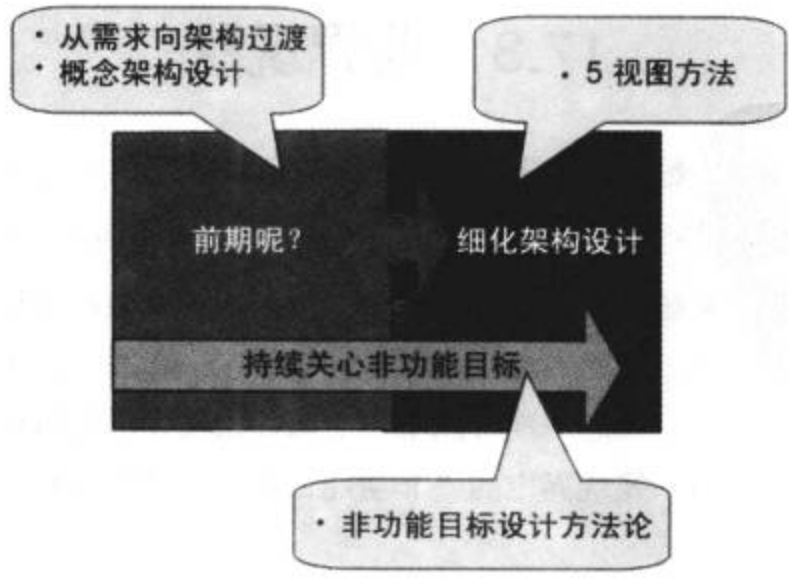


图 17-3　非功能目标的考虑是纵穿架构设计始终的环节

第 18 章
方法："目标-场景-决策"表

软件的质量是设计出来的，这是公认的一个基本观点。

——邓成飞，《软件工程管理》

很少有需求文档能够以一种可测试的细节捕获系统所有的质量需求。现实情况是设计师通常不得不填补空白并协调冲突。

——Len Bass，《软件构架实践（第 2 版）》

核心竞争力是什么？答案是：能力的稀缺性。

本章介绍的目标-场景-决策表方法，可以帮助架构师快速建立非功能目标的设计思维，更理性地应对架构师普遍感到棘手的非功能支持问题，提升自己的核心竞争力。

18.1 场景技术

18.1.1 场景技术的历史

我们用图 18-1 来浓缩场景技术的历史。

- **诞生**
  二战后，美国空军用场景技术想象对手会采取哪些措施，然后准备相应的战略。
- **转变**
  196x 年，兰德公司和曾供职于美国空军的赫尔曼·卡恩，将这种军事规划方法提炼成一种商业预测工具。
- **成名**
  壳牌公司运用它成功预测了 1973 年的石油危机，而名声鹊起。（《福布斯》杂志 1970 年还称壳牌公司为"丑美"，但后来……）。
- **应用**
  据贝恩公司 2004 年对 960 家跨国公司经理的调查表明，场景技术应用超过 50% 。

图 18-1 场景技术的大致历史

于是，许多同行困惑的"场景技术书籍难找"的问题也有了答案——由于场景技术并不产生于计算机行业，而且如今商业领域对场景技术的重视大大超过了软件行业，所以商业书店里"场景规划"方面的书籍是我们的有益参考。

18.1.2　软件行业中场景技术的应用现状与展望

在软件行业，场景技术有着广泛的应用，并且未来场景技术将更加重要。

先说用例。很多软件从业者知道场景，都是从用例开始的。值得说明的有几点：

- 有书上说，场景就是用例，此观点错误。
- 用例是能给外部角色带来可见价值的交互序列，1 个用例 ＝N 个场景。
- 用例是功能需求实际上的标准，它不能全面涵盖系统的非功能需求。
- 场景既可以是功能场景，也可以是非功能场景。

再考虑基于场景的架构评估方法。例如 ATAM 方法，其主要流程的示意图如图 18-2 所示。在此类方法中，场景技术作为一种质疑技术发挥作用——架构对目标的支持怎么样呀？

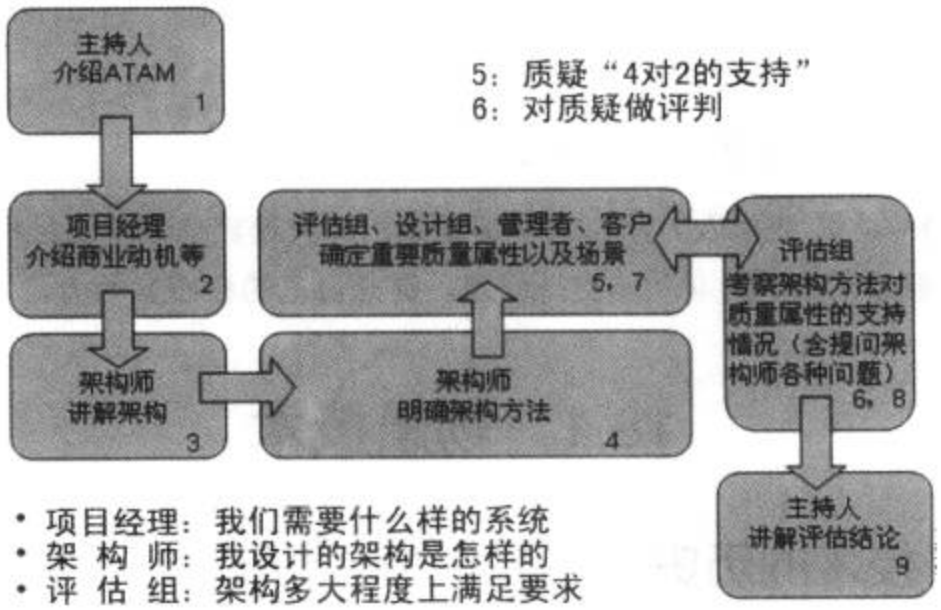


图 18-2　ATAM 方法主要流程示意图

场景的"应用案例"还有很多，例如测试用例、业务需求等（如图 18-3 所示）。

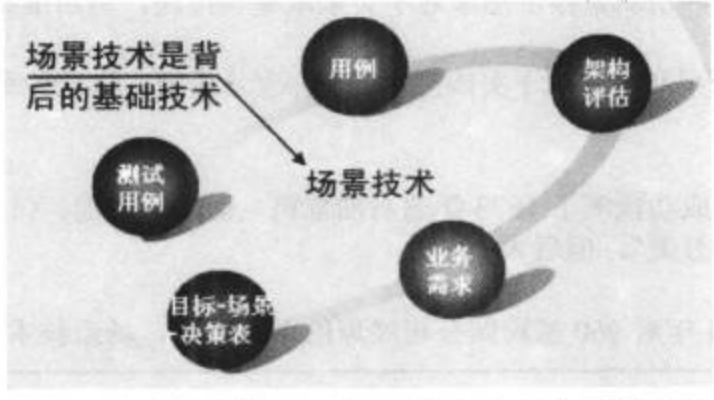


图 18-3　在软件行业，场景技术有着广泛的应用

至此，可得到一个重要的总体结论：

场景是基础技术，用例是应用技术。

为了以结构化的方式描述“能给外部角色带来可见价值的交互序列”，将成功场景和各种意外场景“打包”形成了用例。

于是不难理解，用于测试设计的“测试用例”、用于评审的“基于场景的架构评估方法”、用于支持非功能目标设计方法论的“目标-场景-决策表”……都是场景技术的具体应用。

展望未来，场景技术在软件行业的地位不仅不可撼动，而且会愈加关键。例如，软件行业至今存在的核心问题之一是“需求的可验证性”问题，它是造成眼下许多困难的重要根源：

- 测试的流程经常是：测试需求、测试设计、测试实施……暂停！我们发现这里有个“测试需求”环节，为什么又投入精力做“需求”工作呢？
- 架构设计中也反映了类似的问题，架构师面对“高性能”这样“需求可验证性”很差的非功能需求无法有效展开设计。
- ……

为此，软件行业未来在需求领域必然会不断发展：

- 通过场景化，增强非功能需求的可验证性（那时候，业界公认的软件工程过程中描述需求分析师的工作时，会出现类似本书“目标-场景-决策表”的很多工作……）。
- 场景技术和“定量的需求”日渐结合，发挥两个技术各自的优点——本质上，场景和定量是正交的、相互独立的两种技术（如图 18-4 所示），它们的“珠联璧合”是必然趋势。

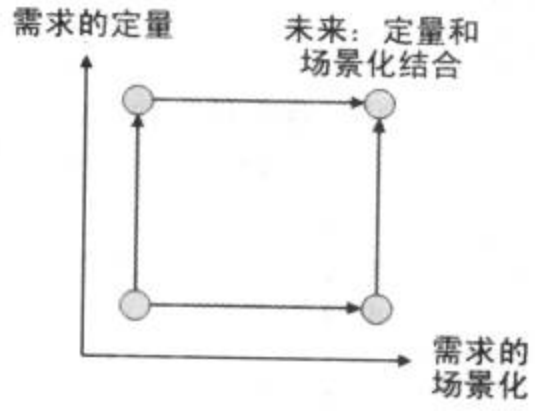


图 18-4 需求的可验证性：场景和定量两种技术的结合是必然趋势

18.1.3 场景的 5 要素与场景卡

和其他文献不同，本书建议场景应包含 5 要素：

- 影响来源。来自系统外部或系统内部的触发因素。
- 如何影响。影响来源施加了什么影响。

- 受影响对象。默认为"本系统"。
- 问题或价值。受影响对象因此出现什么问题，或者需要体现什么价值。
- 所处环境。此时，所处的环境或上下文为何（此要素为可选要素）。

了解了场景的5要素，场景卡就非常容易理解了：一张以5要素为核心内容的需求采集卡片。当需求分析师并未通过场景技术明确定义非功能需求，当架构师也深感难以到位地发现有价值的场景时，架构师可以借助场景卡来激活团队的力量——让大家提交场景。例如，图18-5就展示了一个填写了内容的场景卡。

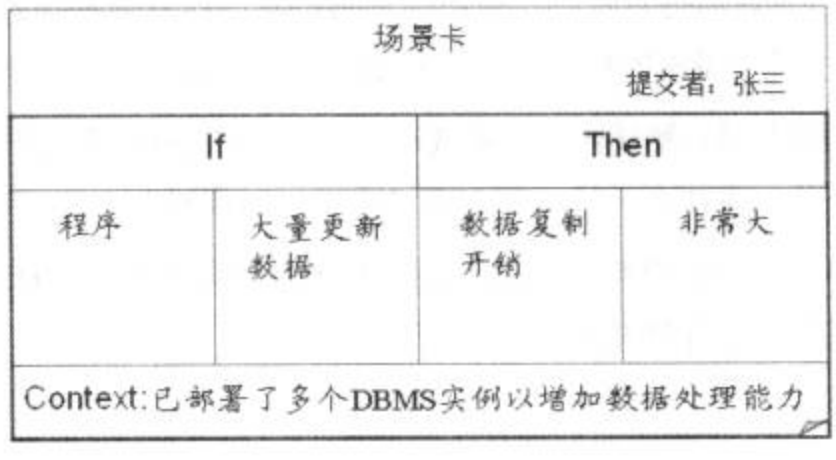


图18-5　场景卡一例

18.2　"目标-场景-决策"表

软件界泰斗Edsger Dijkstra指出，"我们所使用的工具深刻地影响我们的思考习惯,从而也影响了我们的思考能力"。如表18-1所示，目标-场景-决策表将思考过程形象化、可视化了，能够非常务实有效地促进架构设计思维。

表18-1　目标-场景-决策表

目标	场景	决策
业务需求和约束 细化	业务场景	架构决策
	约束场景 支持	架构决策
	约束场景	架构决策
质量属性 细化	质量场景	架构决策
	质量场景 支持	架构决策
	质量场景	架构决策
质量属性 细化	质量场景	架构决策
	质量场景 支持	架构决策
	质量场景	架构决策

实践中，架构师必须对场景进行评估，以决定是否支持这个场景。架构师经常要考虑的场景评估因素包括：

- 价值大小。

- 代价大小。
- 开发难度高低。
- 技术趋势。
- 出现几率。

最后，必须提醒的是“不支持某场景”恰恰是架构师的一种重要决策——如果每个场景都给予支持，理性设计就无从谈起，多度设计就在所难免了。如表 18-2 所示的目标-场景-决策表中，就明确决定：不支持掌上电脑作为项目管理系统的客户端。

表 18-2 项目管理系统：如何支持非功能目标的思考过程

目标	场景	决策
业务需求和约束	• 项目成员，在客户现场，能够访问 PM 系统	• B/S 架构
	• PM 系统的用户，分别在 windows、Unix、Linux、Mac 上工作，能够访问 PM 系统	• B/S 架构
互操作性（质量属性）	• HR 系统，已有员工信息，不再输入而是互操作导入	• 支持从 HR 系统导入员工信息功能
	• PM 系统的用户，也要管理项目文档，同时访问 PM、CVS 麻烦	• PM 作为 CVS 客户端提供文档管理能力
	• 配置管理服务器，不是 CVS，如何支持	• Adapter 设计模式，支持 CVS、Subversion 等
跨平台（质量属性）	• DBMS，可能是 Oracle、SQL Server，如何都支持	• 采用 ORM 技术
	• 客户端，运行在掌上电脑上，如何支持	• 不支持
	• 服务器，跑的 OS 可能是 Windows、Unix、Linux，如何都支持	• Java 开发 Server 端

索引
index

编辑手记

温昱老师是博文视点的老朋友，他在博文视点出版的译作《应用》及著作《软件架构设计》都是深受读者好评的高质量图书。今年8月，周老师让我负责温昱老师的新书《一线架构师实践指南》，我心有惶恐，很怕做不好这项工作。

作为编辑，专业深度肯定及不上作者，但武汉博文视点对于原创技术图书，有严格的专家审稿制度。在周老师的指导下，我邀请了一批资深的架构师审阅书稿。参与审稿的专家包括：支付宝首席架构师程立、支付宝业务架构师周爱民、豆瓣网技术总监洪强宁、微软亚洲研究院技术创新组研发经理邹欣、支付宝数据架构师冯大辉、手机之家网站创始人高春辉。专家们从各自的技术视角及关注点出发，分别对书稿的目录结构、观点、方法，以及内容表述提出了修改建议。这些建议大部分都被温昱老师采纳在书稿中。而参与组织专家审稿，也是编辑能快速提升自己对书稿内容的理解的极好机会。

为了进一步理解书稿的内容，在请专家审阅书稿的同时，我仔细地阅读了温昱老师的第一本著作《软件架构设计》一书。这本书为我这个架构设计技术的门外汉清晰地描绘了软件架构设计的全貌，让我对需求分析、领域建模和架构视图等概念有了更深入更全面的理解，我还在周老师的建议下，阅读了温老师的译作《应用》一书，两本专业书的持续阅读（我坚持每天写阅读心得），也让我为今后策划架构设计相关的选题增添了信心。

普通读者阅读温老师的书得自己掏钱，编辑读温老师的书不用花钱，在此向温老师表示衷心的感谢！

软件架构应当为开发人员提供足够的指导和限制，但是分析软件架构设计的书不可能写成具体项目的架构说明书，否则针对性太强，普适性就太差了。它必然要总结不同业务领域中架构设计的规律，提炼出有共性的方法和思路。然而总结规律就需要有很强的抽象能力，所以有读者说《软件架构设计》有些“高来高去”，指导性不强，而这本《一线架构师实践指南》则具有深入浅出的意味，希望它的出版，能满足这些读者的期待。

——策划编辑 徐定翔

秋风起，竹枝摇曳。温昱老师的《一线架构师实践指南》就要面市了。

架构，原先一直觉得这个词很神秘，而架构师也是一个很“牛”的职业，让人高山仰止。当我认真阅读完全书后，竟然发现自己这个架构盲也颇有心得，架构师身上的神秘面纱被温昱老师以幽默的笔调、深入浅出的讲解、贯穿的实例演示给揭开了。和徐定翔编辑相比，我对技术更缺乏认识和了解，但温老师的书，我能看懂大半。

我很佩服徐定翔编辑的，还有他那种和作者沟通的韧劲及灵活劲儿，到后来看到温昱老师在邮件中写道：“定翔，我对你的尊敬有增无减……” 我开心地笑了，这就是编辑工作中的乐趣所在。

作为责编，我花了不少功夫来一遍又一遍地和温昱老师沟通书中图片和表格的处理，为一幅图的修改，为一张表的调整颇费斟酌，但当制作出理想的效果时，那种快乐也是很难替代的。

秋天是收获的季节，愿温昱老师和我们的劳动能让读者品尝到果实的芳香。

——责任编辑 白爱萍

本书豆瓣：
http://www.douban.com/subject/4031899/

博文视点（武汉）豆瓣：
http://www.douban.com/people/broadview/

竹影婆娑

——《一线架构师实践指南》设计手记

听编辑说，作者温昱老师清瘦飘逸，仙风道骨，颇具“竹”韵。

不断听编辑讲和温老师沟通改稿的故事，感受到温老师有股竹子才有的韧劲。

起初，温老师对第一稿的设计方案提出了较多意见，逐条阐述了他的想法和理由，甚至还自己动手修改封面，更煞费苦心找来不少资料供我们参考。我当时还有点担心这样的调整会影响到整个系列的风格，后来与温老师多次沟通磨合，最终第三稿顺利得到他的认可。

封面定稿后，收到温老师发来的致谢信，让人感到温暖，受到鼓舞。他的谦和，让我想起了竹子的虚心，真是可爱又可敬的人。

画竹之高手莫过于郑板桥，查看了不少板桥先生的竹画资料后，最终我还是没有采用他的画作。作为“八怪”的代表人物，板桥先生一生痛恨官场腐败，他所画的竹透露出一股凌厉的抗争之气，充满了艺术家的狂放不羁。这种绘画风格对于本书而言还是过于沉重了。

“宁可食无肉，不可居无竹”，我很羡慕古人对竹子的钟爱。秋风渐起，竹影婆娑，这里借用板桥先生的《竹石》诗来与大家共同体会竹之精神：

咬定青山不放松，立根原在破岩中；

千磨万击还坚劲，任尔东西南北风。

——杨小勤

书稿中的原始图片不少是适宜在网络上使用的彩色图片，色彩鲜艳，易于识别。可用到书上只能用黑白灰三色来区分。考虑到这样的印刷限制，书中大量的图表都由设计部重新制作，甚至重新设计，设计师为此反复检验修改后的效果，有时候一张图需要修改上 10 次才能达到要求。全书的图表处理下来，耗时不少。但能使作者的信息更准确更清晰地传达给读者，还是颇有成就感的。祝大家阅读愉快。

——胡文佳　曾绪凡

博文视点 重磅推荐

《编程之美》

★荣获中国书刊发行业协会组织的
“2008年度全行业优秀畅销品种”奖

★荣获CSDN评选的“软件中国2008图书类编辑选择奖”

★51CTO读书频道和中国图书商报、中国互动出版网共同评选为
“2008年度最佳技术图书”

充满智慧与趣味、囊括大量有趣且有启发性的面试题目
让您充分享受思考之乐、编程之美

这本书表面上是讲解算法，实际上体现了一种面对困难、解决问题的心态……个人还是挺喜欢这类书的，把编程人性化了……

——拓荒者

《编程之美》中的算法以实例开题，循序渐进地解决问题，一步步去剖析算法的本质，挖掘和发散算法功效，进而去淋漓尽致地体现算法的美妙！

——萝卜萝卜闪金光

《程序员的自我修养》

俞甲子、石凡、潘爱民原创精品

深入浅出地对系统软件的底层形成机制进行条分缕析

了解程序的前世今生，

彻底理解敲入的代码如何变成程序

关于链接、装载等问题，是操作系统中很基础很重要的一个部分……一方面，我们理解系统如何去做，是为了悟到为何这样去设计。了解了为什么，反过来更能理解怎样去做……另一方面，我们对自己每天用的系统多一些了解，那是百利而无一害。

——云风

博文视点软件工程

《代码大全（第2版）》

[美]Steve McConnell 著
金戈 汤凌 陈硕 张菲 译

- 两届震撼大奖得主，集数十年软件开发智慧
- 年销售逾30000册

《精益软件开发艺术》

Curt Hibbs, Steve Jewett & Mike Sullivan 著
章显洲 译

- 揭秘源自日本工业的精益方法
- 软件领域实施“精益生产”的导航图
- 简明扼要 一语中的

《Visual Studio Team System 更佳敏捷软件开发》

Will Stott James Newkirk 著
刘志杰 译

- 为在现实敏捷开发环境中实现VSTS提供全面、透彻指导
- 展示如何用VSTS逐步改善软件开发的方方面面
- 帮助读者轻松理解并顺利掌握敏捷实践

《大道至简——软件工程实践者的思想》

周爱民 著

- 从工程实践出发溯源而论的国内原创佳作
- 深入讨论软件工程思想的本源

《设计模式》

王翔 著

- 全国海关中心架构师《程序员》专栏作者王翔倾力奉献
- 深挖C#语言高级特性
- 以简洁、直接的手段解决易变化的问题

《移山之道——VSTS软件开发指南》

《移山之道——VSTS软件开发指南（第2版）》

邹欣 著

- 第一本由微软Visual Studio Team System一线开发人员所写的原创精品
- 第一本直接针对中国软件开发人员写的循序渐进的、基于案例的教材
- 第一本通过微软自身的实践直接介绍微软开发流程（MSF）的实用手册

《Windows用户态程序高效排错》

熊力 著

- 深入剖析数十个微软企业客户的真实案例
- 让您成为福尔摩斯一样的排错高手

《软件调试》

张银奎 著

- 全方位展示调试技术的威力和魅力
- Jack B. Dennis教授撰文支持
- David A. Solomon作序推荐

《程序员修炼之道——从小工到专家》

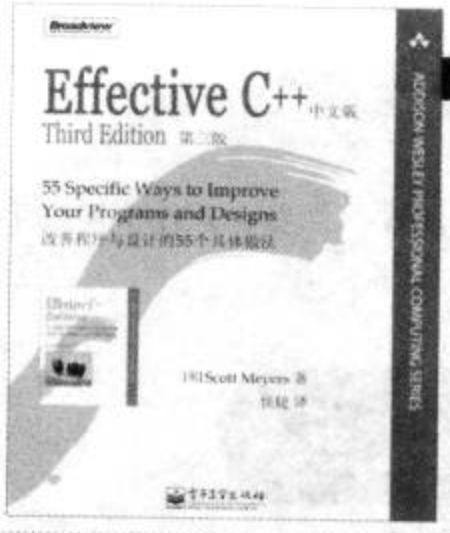
Andrew Hunt　David Thomas　著
马维达　译

- 全面阐释软件开发不同方面的最佳实践和重大陷阱
- 网易、趋势科技等公司新员工技术培训首选用书

《UNIX编程艺术》

[美]Eric S. Raymond　著
姜宏　何源　蔡晓骏　译

- Addison-Wesley旗舰系列之“最高品质”
- 它不是技术方面的单纯介绍，而是编程哲学、编程思想。编程文化、编程艺术的全面讨论，将带给您十分愉悦的感受



《Effective C++中文版，第三版》

[美]Scott Meyers　著
侯捷　译

- 自1991年《Effective C++》第一版之后，《Effective C++》第三版隆重出版
- 目标：在一本小而有趣的书中确认最重要的一些C++编程准则
- 作者Scott Meyers以一支生花妙笔将复杂的探索过程和前因后果写成环环相扣、故事性强的文字

《深入解析Windows操作系统 第4版——Microsoft Windows Server 2003/Windows XP/Windows 2000技术内幕》

[美] Mark E. Russinovich, David A. Solomon　著
潘爱民　译

- 微软官方权威参考书，名著名译！
- Windows系统之父Jim Allchin、Windows Nt首席设计师David N. Cutler 写序推荐！
- CSDN、《程序员》杂志、博客园、博客堂鼎力推荐！

Broadview®

北京博文视点（www.broadview.com.cn）资讯有限公司成立于2003年，是工业和信息化部直属的中央一级科技与教育出版社——电子工业出版社（PHEI）下属旗舰级子公司，在六年的开拓、探索和成长中，已成为中国颇具影响力的专业IT图书策划和服务提供商。

六年来，博文视点以开发IT类图书选题为主业，励精图治、兢兢业业，打造了一支团结一心的专业队伍，并形成了自身独特的竞争优势。一直以来，博文视点始终以传播完美知识为己任，用诚挚之心奉献精品佳作，年组织策划图书达300个品种，同时开展相关信息和知识增值服务，赢得了众多才华横溢的作者朋友和肝胆相照的合作伙伴，已经成为IT图书领域的高端品牌。

我们的理念：创新专业图书服务体制；培养职业策划图书服务队伍；打造精品图书品牌；完善全面出版服务平台。

我们的目标：面向IT专业人员的出版物提供相关服务。

我们的团队：一个整合了专业技术人员和专业服务人员的团队；一个充满创新意识和创作激情的团队；一个不断进取、追求卓越的团队。

我们的服务：善待作者　尊重作者　提升作者

我们的实力：优秀的专业编辑队伍
全方位立体化的强大的市场推广平台
实力雄厚的电子工业出版社的渠道平台

“走出软件作坊独辟蹊径　人道编程之美，
追踪加密解密庖丁解牛　精雕夜读天书。”

路漫漫其修远，博文视点愿与所有曾经帮助、关心过我们的朋友、作者、合作伙伴携手奋斗。未来之路，不可限量！

博文视点

地址：北京市万寿路173信箱电子工业出版社博文视点资讯有限公司
邮编：100036　　**总机**：010-88254356　　**传真**：010-88254302

武汉分部地址：武汉市洪山区吴家湾湖北信息产业科技大厦1402室
邮编：430074　**总机**：027-87690813　**传真**：027-87690013

欢迎投稿：bvtougao@gmail. com
读者邮箱：reader@broadview.com.cn
博文视点官方博客：http://blog.csdn.net/bvbook/
博文视点官方网站：http://www.broadiew.com.cn/

在线有奖读者调查表

Broadview®
www.broadview.com.cn

《一线架构师实践指南》

http://blog.csdn.net/bvbook

登录以上网站告诉我们您关于这本书的建议、意见
就有机会获赠博文视点的『**新书一本**』
并参加年终大抽奖活动

您的支持就是我们创造精品动力的源泉!

欢迎投稿: bvtougao@gmail.com

读者信箱: reader@broadview.com.cn

博文视点更多资源网站:

- VSTS虚拟社区:
  http://yishan.cc/
- 《代码大全》资源网站:
  http://www.cc2e.com.cn/
- 博文视点其他博客:
  http://www.cnblogs.com/bvbook/
  http://bvbook.javaeye.com/

反侵权盗版声明