一线架构师实践指南

专家评荐版

温昱 著

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”才能堪此重任。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。

——温昱（shanghaiwenyu@163.com）

目　录

第 2 部分 Conceptual Architecture 阶段

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。

第 1 节　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”、和“2 个职业发展的困惑”。

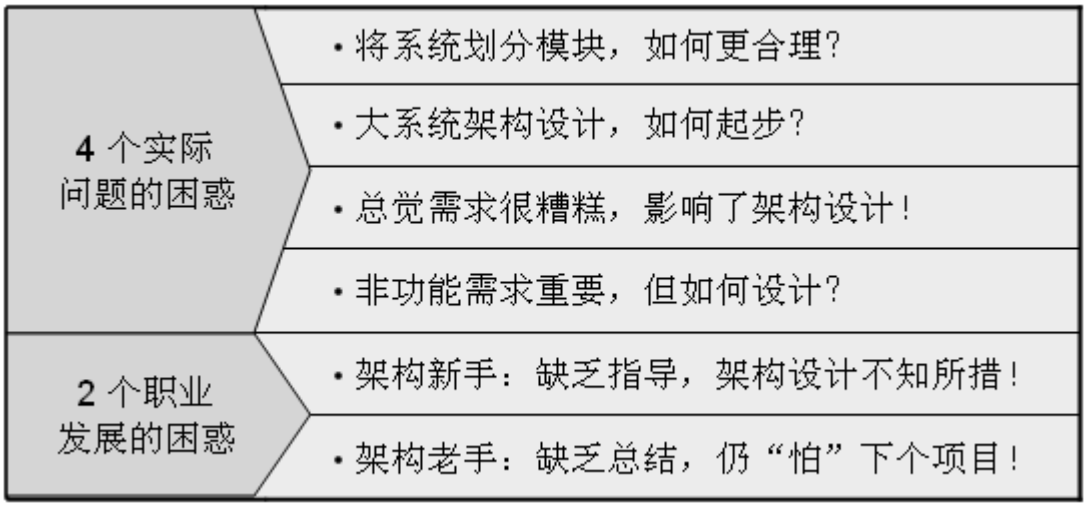


图 1-1　一线架构师的 6 个经典困惑

第 2 节　本书：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

透过这 4 个核心主张，便于读者领会 ADMEMS 方法之精髓。

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”才能堪此重任。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。

2.2 质疑驱动的架构设计

毫无疑问，架构设计是需求驱动的，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”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），有节奏地引入到架构设计的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：这个架构设计中间成果，还需要“我”进一步通过“质疑”引入更多“质量属性”以及“特殊功能场景”来驱动后续的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真正驱动力。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”这种观点，则有严重缺陷，3 句话足以揭示这一点：

- 需求 = 功能 + 质量 + 约束
- 用例是功能需求实际上的标准
- 用例涉及、但不涵盖非功能需求

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。

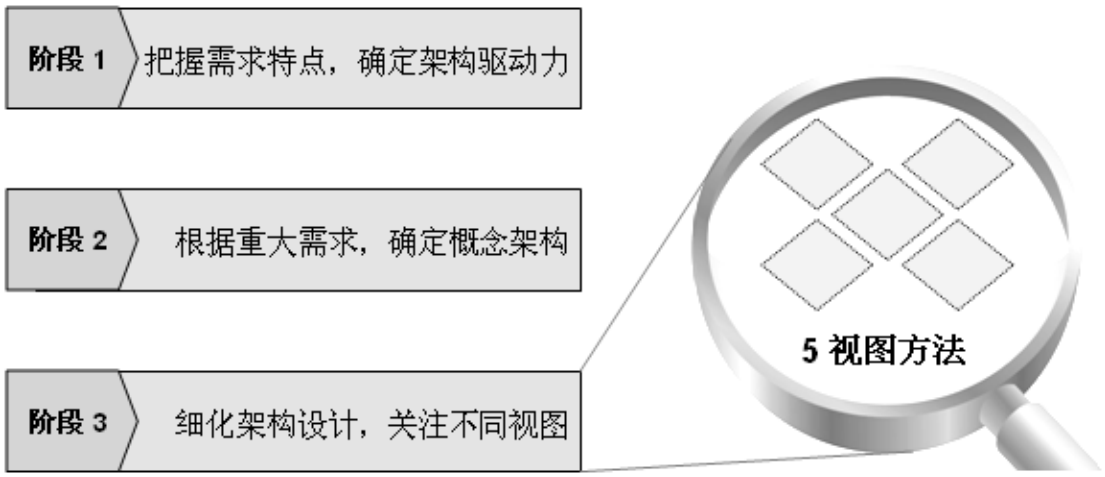


图 1-2　架构设计方法首先应当是多阶段的，其次才是多视图的

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》，其中涉及到架构的内容。
- **另一方面**，团队开发时，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需要做了。
- **结论**，概念架构设计和细化架构设计，是 2 个架构阶段，不是 2 个架构视图。

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
- ……

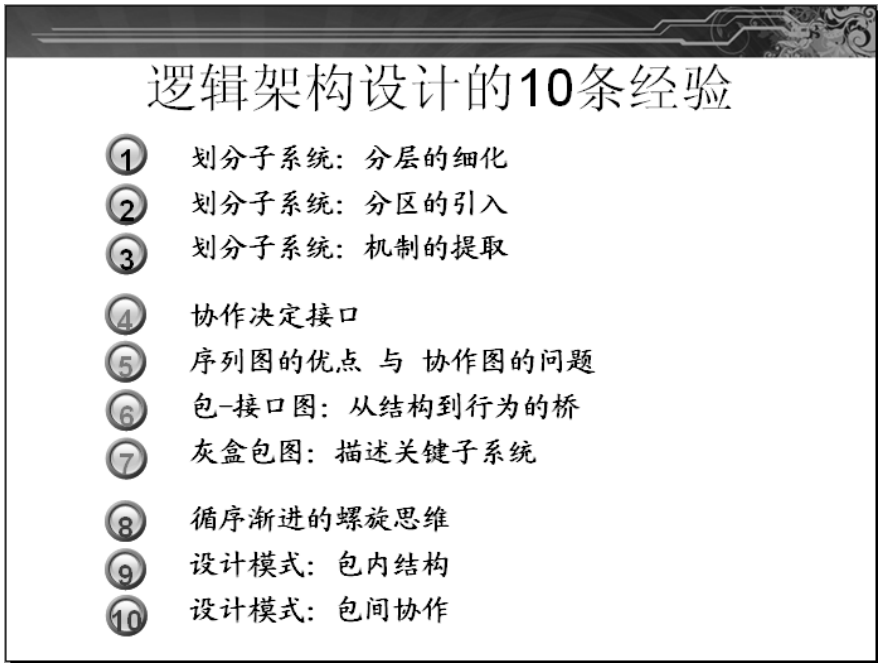


图 1-3　逻辑架构设计的 10 条经验

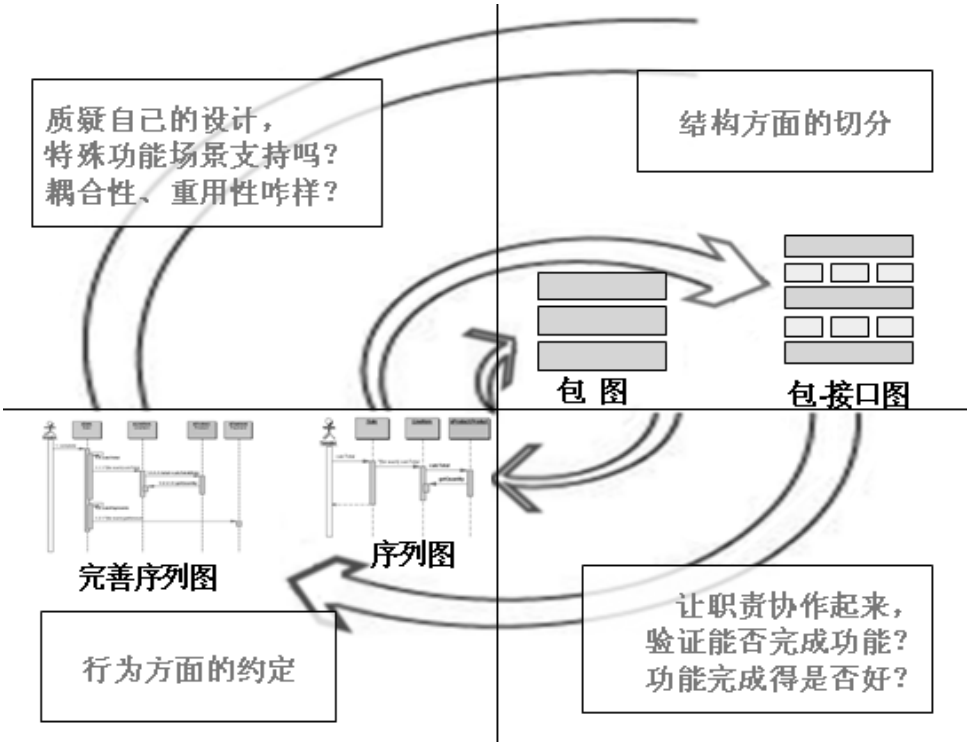


图 1-4　质疑驱动的逻辑架构设计整体思路

第 3 节　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：

- 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
- 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
- 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

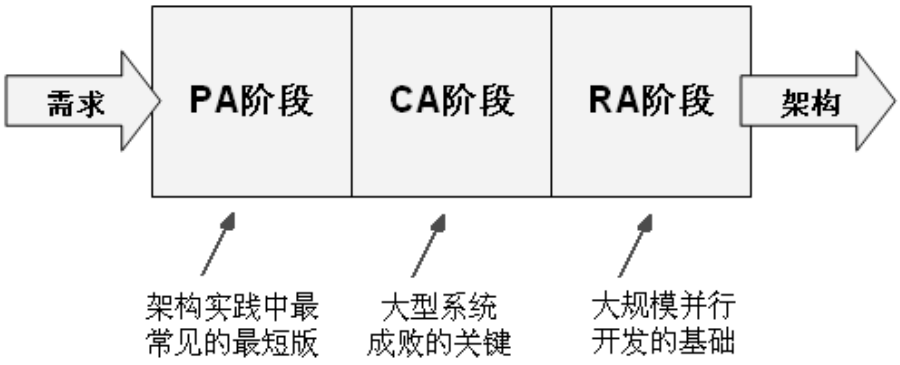


图 1-5　ADMEMS 方法体系包含 3 个阶段

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”，指的是非功能目标的考虑。

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”（如图 1-6 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，藉此避免遗漏需求、并进一步理清需求间关系和发现衍生需求。

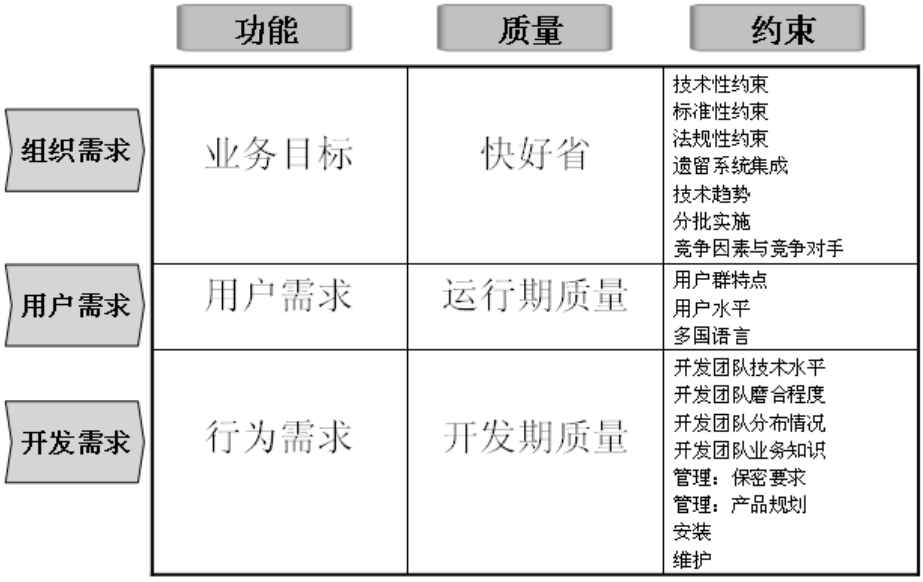


图 1-6　　ADMEMS 矩阵的基础是二维需求观

3.2 Conceptual Arch 阶段：重大需求塑造概念架构

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 1-7 说明了 ADMEMS 方法推荐的概念架构设计的高层步骤。

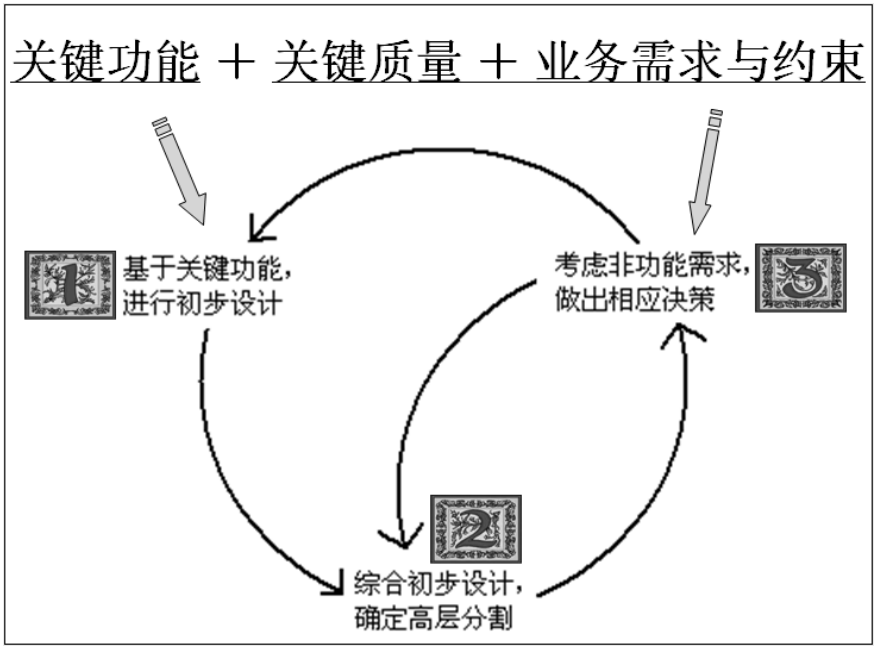


图 1-7　ADMEMS 方法推荐的概念架构设计做法

3.3 Refined Arch 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 1-8 所示。

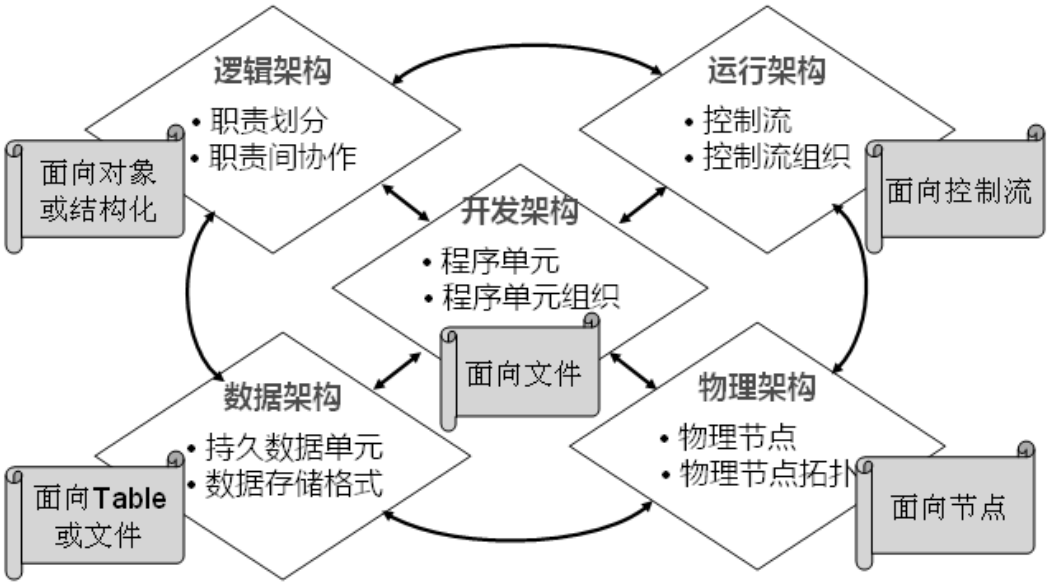


图 1-8　5 视图法：ADMEMS 方法的一部分

许多架构师，言架构则必谈 OO。在他们的思想里边，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 1-8。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，其逻辑架构设计还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”更切近本质。

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-决策表漂亮地将架构师的思维可视化了。例如，如图 1-9 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。

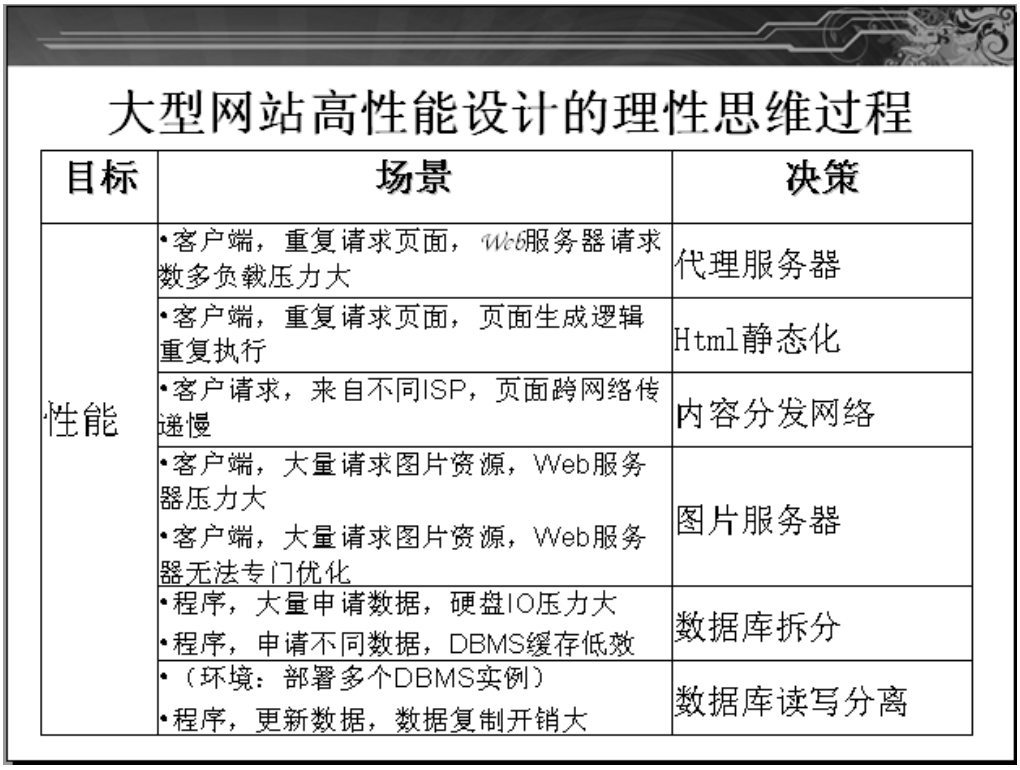


图 1-9　目标-场景-决策表：揭示大型网站高性能设计策略背后的理性思维

第 4 节　如何运用本书解决“6 大困惑”

至此，我们已走马观花地领略了本书所讲的 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 1-10 所示，针对 6 个困惑分别给出了阅读路线图。

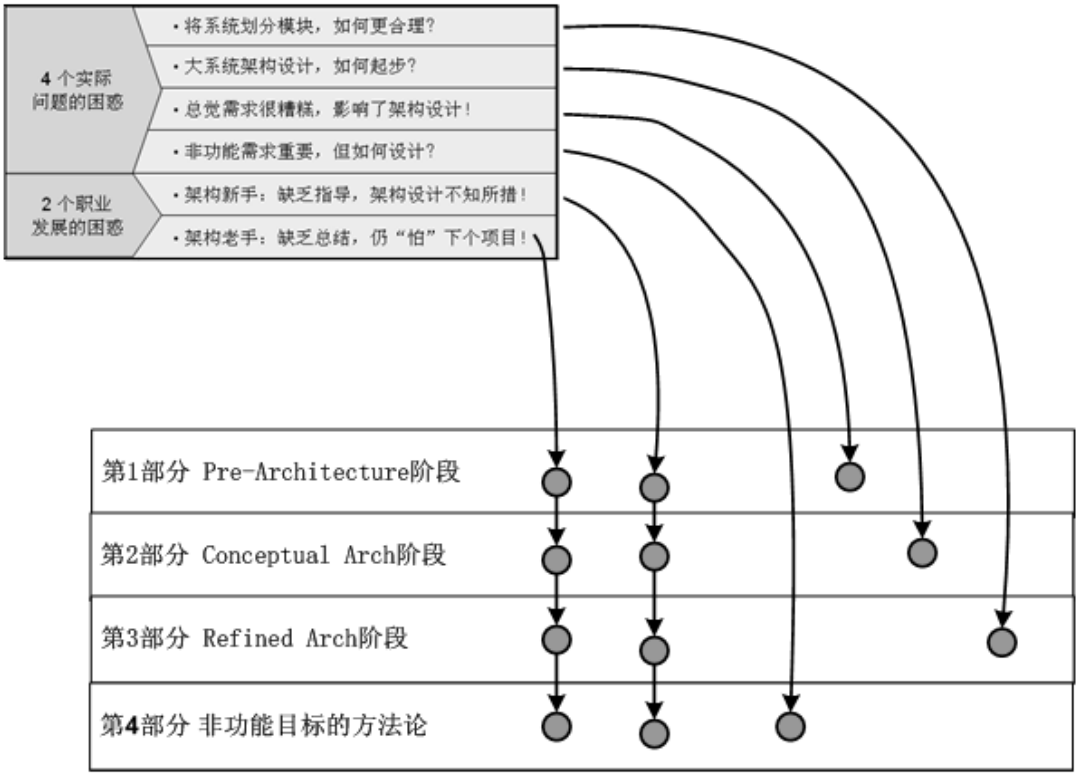


图 1-10　　针对 6 个困惑，不同的阅读路线图

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，应特别关注本书的“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 1-11 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

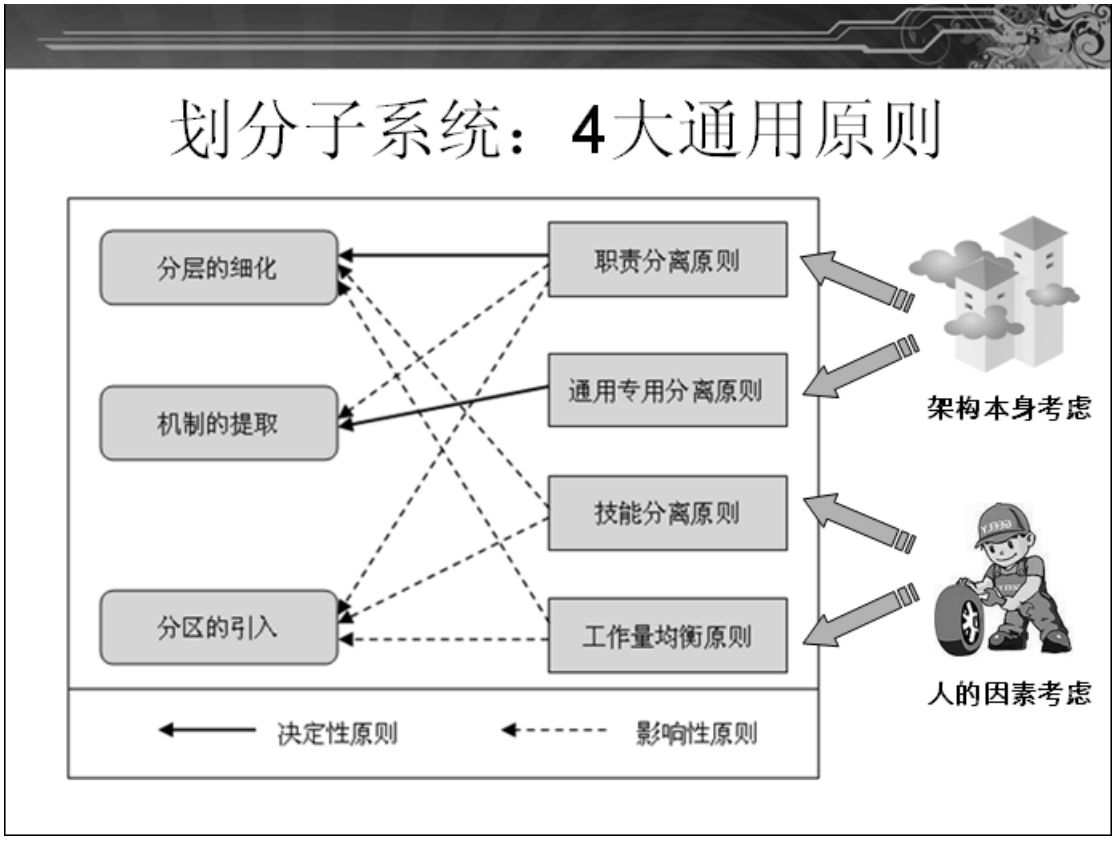


图 1-11　第 3 部分 Refined Arch 阶段：划分子系统的 4 大原则

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。

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。

第 1 节　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“主要运用”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。借此，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 阶段明确要求必须进行需求结构化，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。

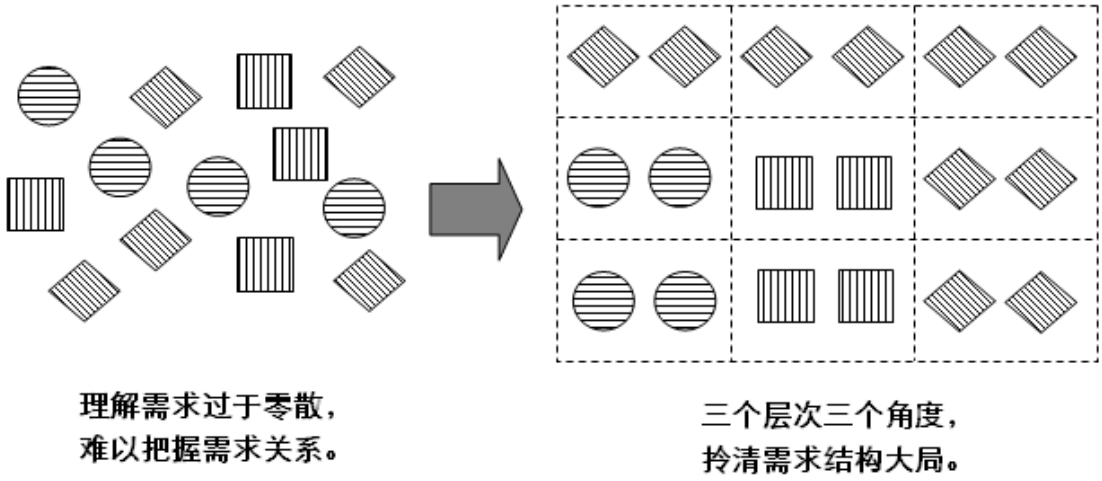


图 4-1　需求结构化的隐喻

第 2 节　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。

2.1 范围：超越《软件需求规格说明书》

不必说，需求文档常常不够全面，所有有经验的架构师都是重视需求文档、但并不“唯需求文档是瞻”。

也不必说，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。

单说最本质的一点，既然架构师必须“对需求进行理性的、有针对性的权衡、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》。

还有一点重大意义在于，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“尽早开始架构设计”一节）。

2.2 工具：ADMEMS 矩阵

矩阵，是很多著名方法的核心。例如，制定公司层战略最流行的方法之一是“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”。如图 4-2 所示。

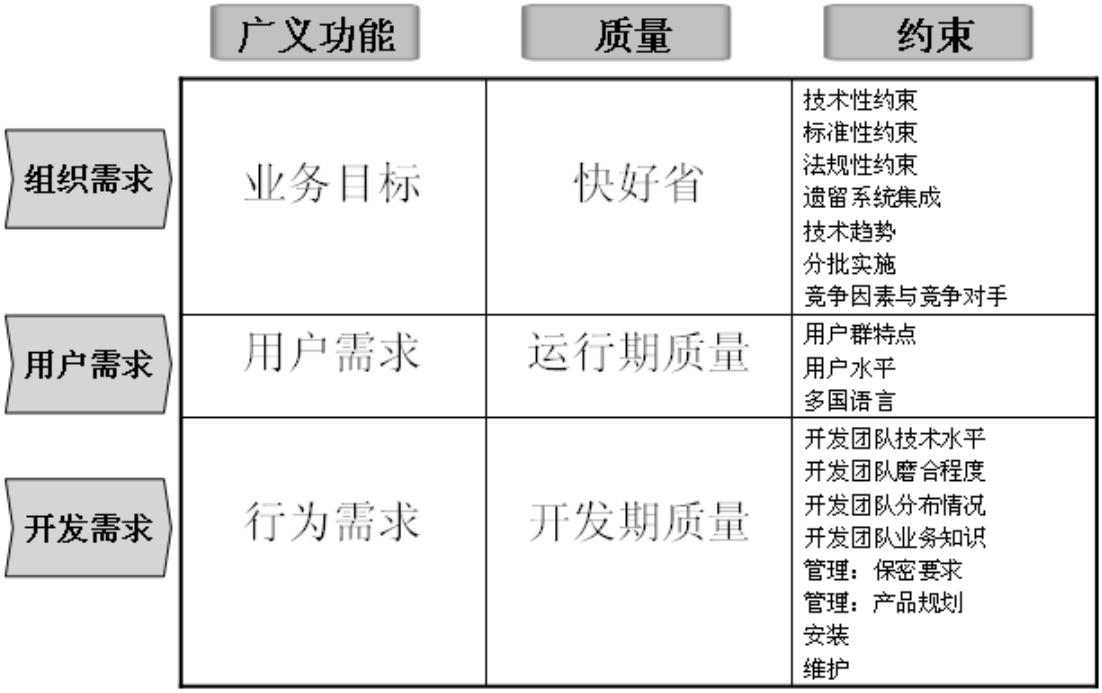


图 4-2　ADMEMS 矩阵（需求层次-需求方面矩阵）

首先，需求是分层次的。一个成功的软件系统，对客户高层而言能够帮助他们达到业务目标，这些目标就是客户高层眼中的需求；对实际使用系统的最终用户而言，系统提供的能力能够辅助他们完成日常工作，这些能力就是最终用户眼中的需求；对开发者而言，有着更多用户没有觉察到的“需求”要实现……

也就是说，从“不同层次的涉众提出需求所站的立场不同”的角度，把需求划分为三种类型，这就是需求的三个层次：

- 业务需求：组织要达到的目标。
- 用户需求：用户使用系统来做什么。
- 行为需求：开发人员需要实现什么。

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，把需求划分为三种类型，这就是需求的三个方面：

- 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不少周折——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 4-3 所示）。

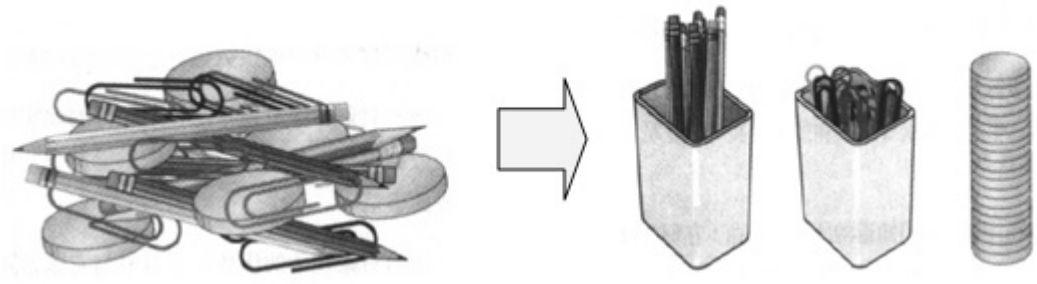


图 4-3　结构化：分门别类地进行“摆放”

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”的一个最大的好处是，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”。

第 3 节　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 4-4 所示的隐喻，点明了分析约束影响的要义：尽早识别风险。

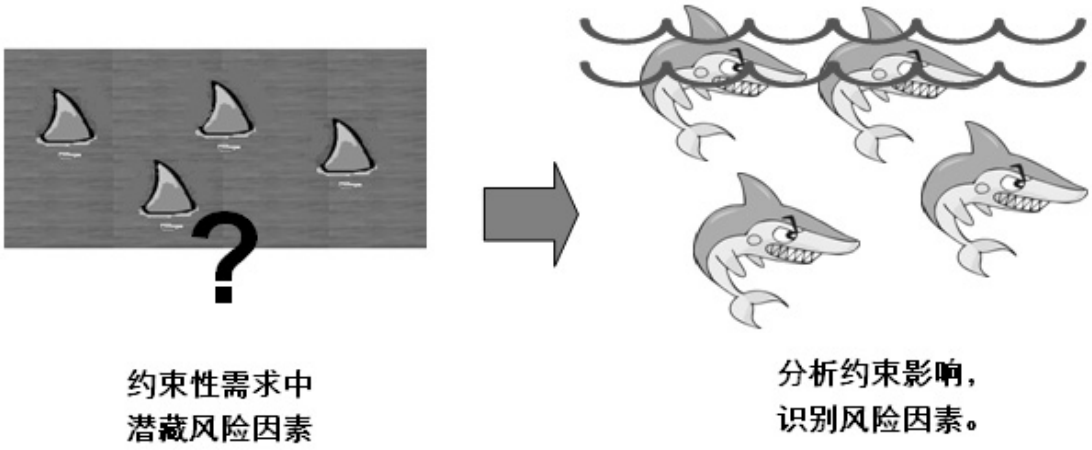


图 4-4　分析约束影响的隐喻

第 4 节　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”。如图 4-5 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组织、用户、开发方这三类涉众，而技术环境则和三类涉众都有关。

图 4-5　4 类约束在 ADMEMS 矩阵中的位置

因为，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，来自客户组织的约束性需求。

- 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为何？有什么业务规则和业务限制？
- 是否需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。

- 开发团队的技术水平如果有限（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，是否需要顾及？
- ……

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。

第 5 节　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 4-6 所示。

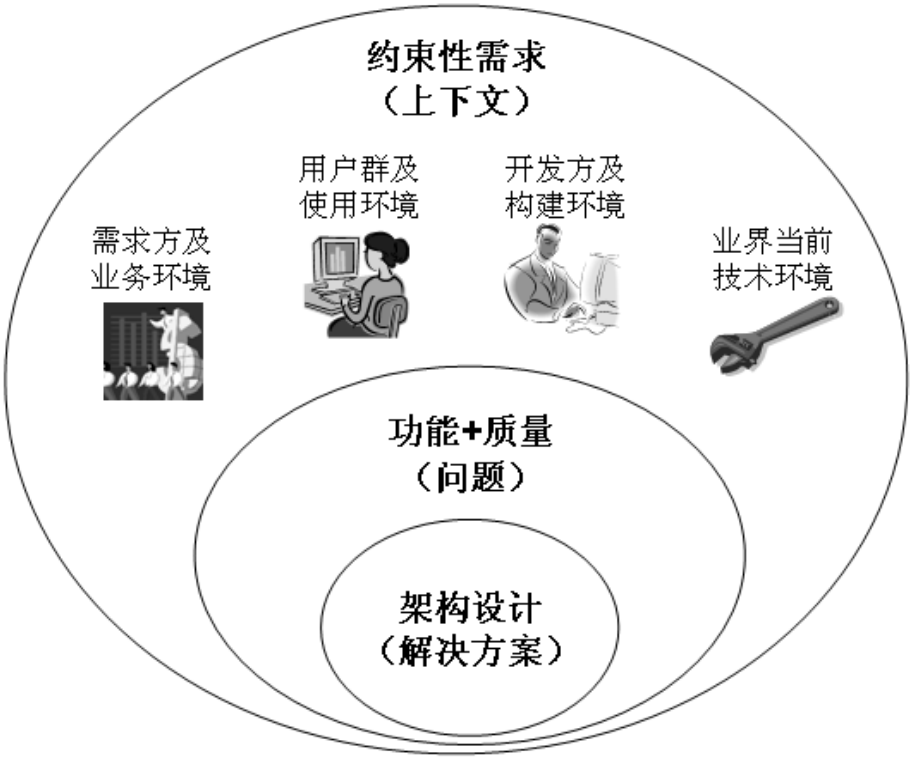


图 4-6　约束是架构设计要解决的问题的上下文，绝对不能忽视

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 4-7 所示，建筑师必须关注以下四类约束的影响，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……

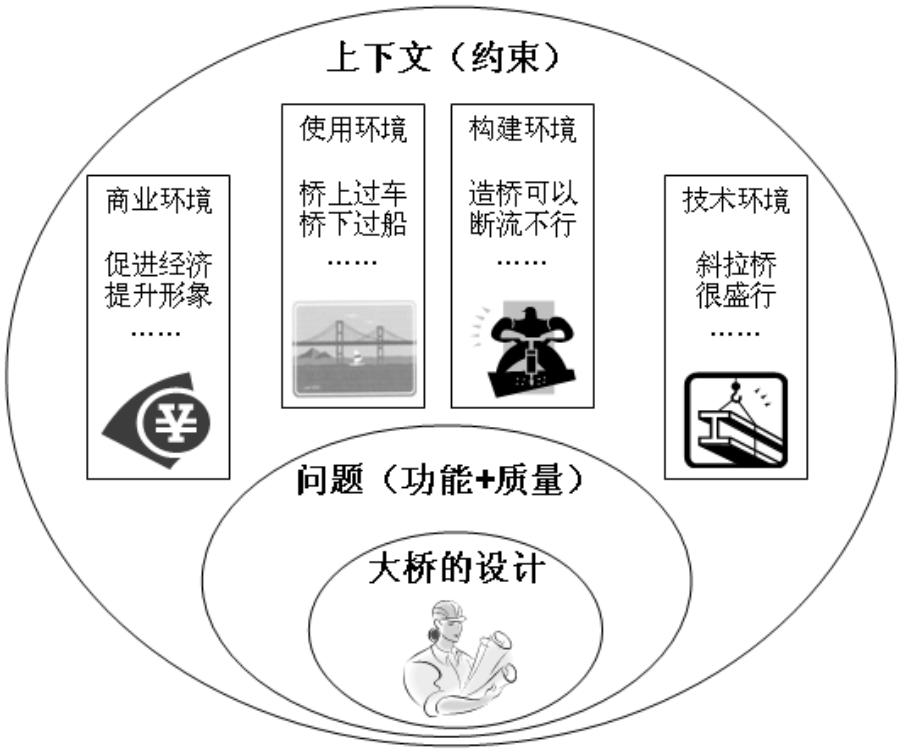


图 4-7 设计大桥：也须考虑“四类约束”的影响

第 6 节　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“四类约束”，并分析约束影响、识别约束背后的衍生需求。

本质上，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二：

- 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

下面，通过案例予以说明。

第 7 节　大型 B2C 网站案例：需求结构化与分析约束影响

像 Amazon 这样的大型 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”。

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。如图 4-8 所示，先来梳理组织一级的需求。

图 4-8　梳理组织一级的需求

用户级需求，要特别注意挖掘来自“用户及使用环境”的约束。如图 4-9 所示。

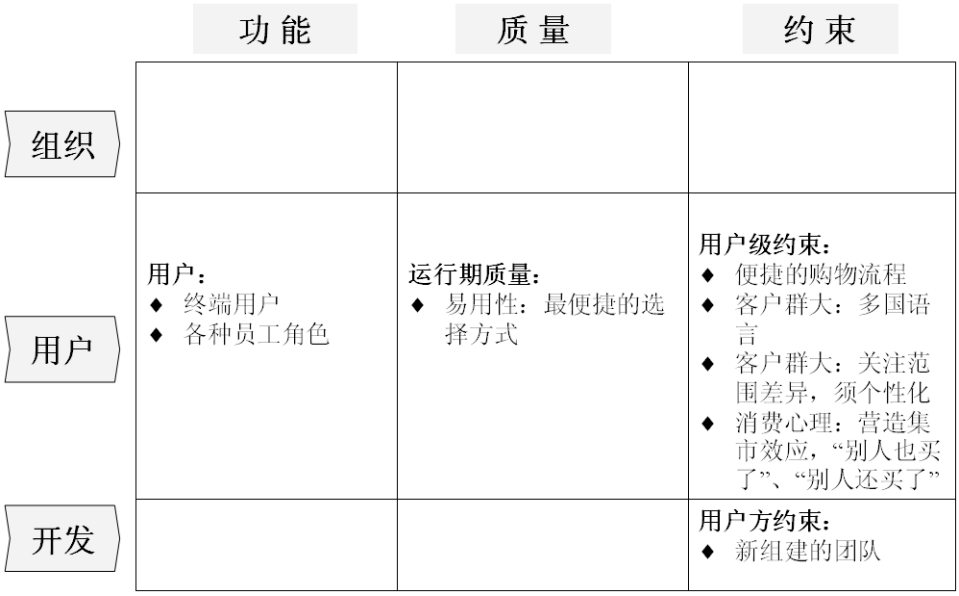


图 4-9　重点的用户级需求

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。如图 4-10 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）、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”。

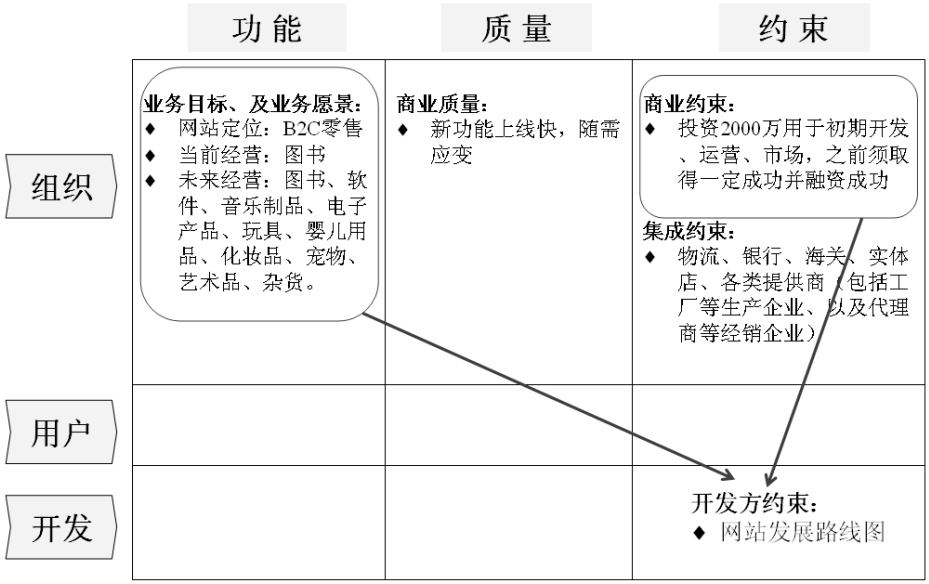


图 4-10　推导法则应用

图 4-11 和 4-12 也是类似思维。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”。

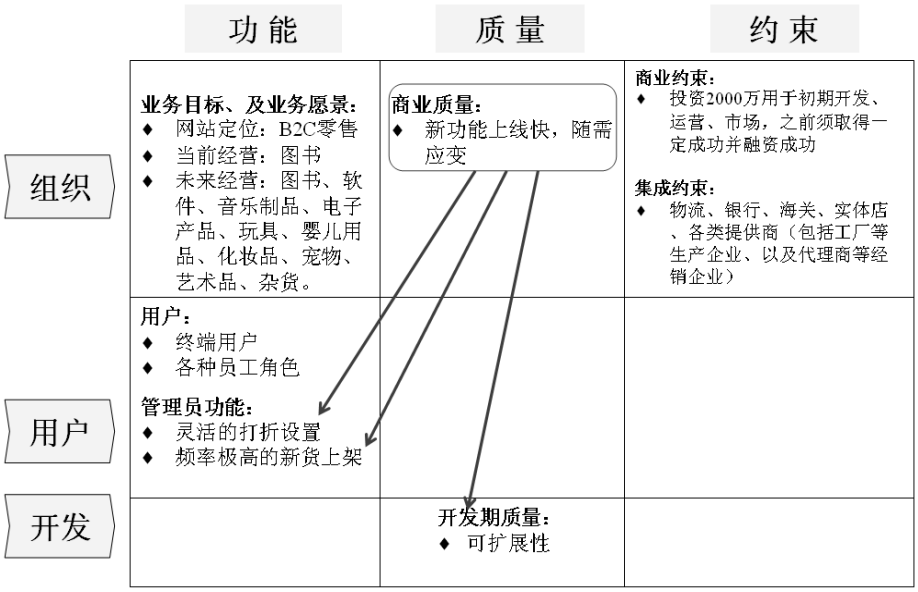


图 4-11　推导法则应用（续）

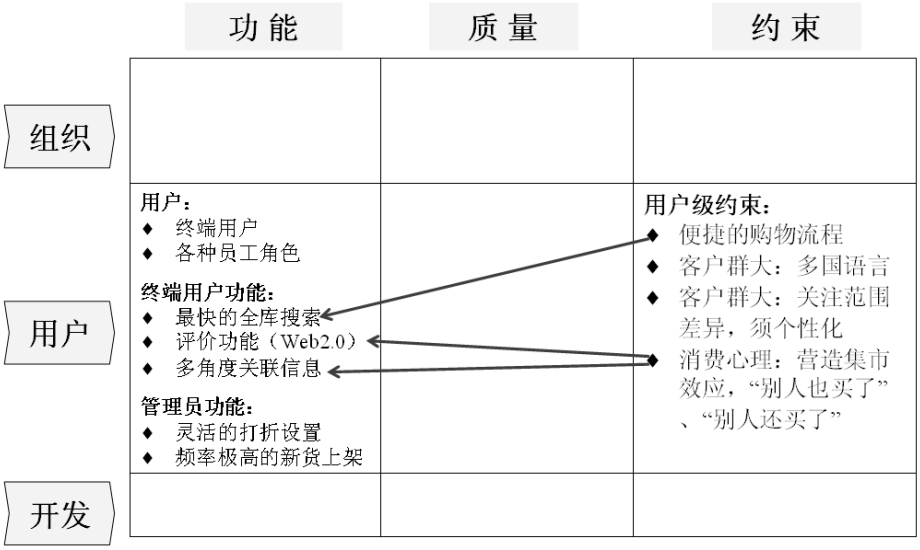


图 4-12　推导法则应用（续）

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？如图 4-13 所示，如此多的集成要求，就必然要提高系统的互操作性……

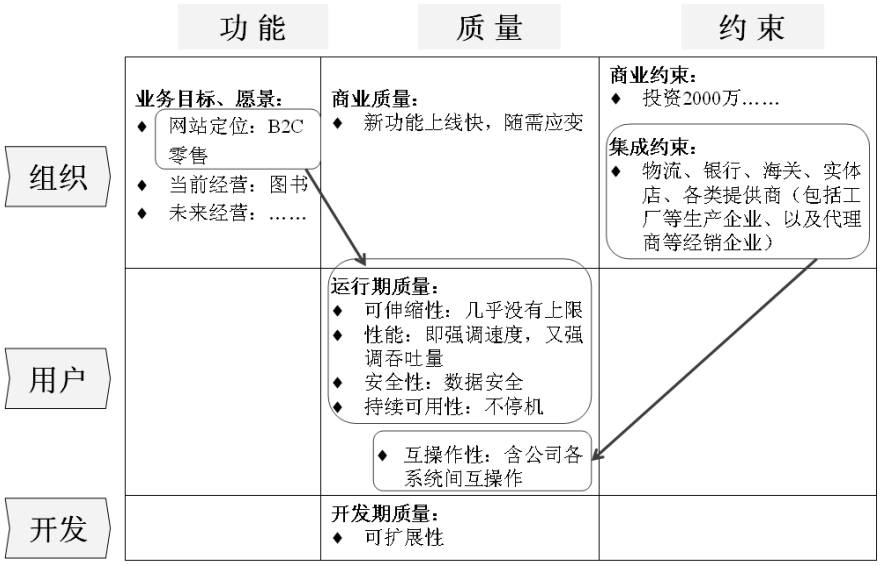


图 4-13　还要主动运用查漏法则

第 8 节　贯穿案例

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。

第 1 节　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，涉及的是你未没接触过的领域，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。初步设计无需展开架构设计细节，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，而分割的目的也恰在规划高层职责模型。

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“不同需求影响架构的不同原理，才是架构设计思维的基础”一节所述，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）。

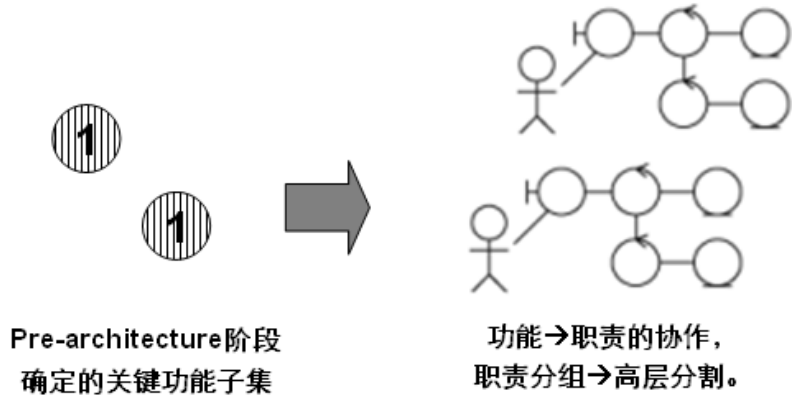


图 8-1　初步设计的具体思路：运用“职责协作链”原理

第 2 节　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？

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。

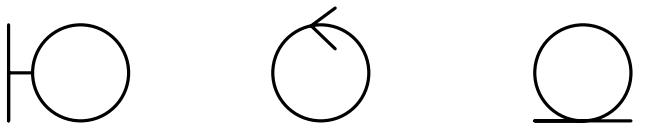


图 8-2　鲁棒图的元素

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。图 8-3 更全面地做了对比，我们发现鲁棒图 3 元素和 MVC 还是有着不小的差异：

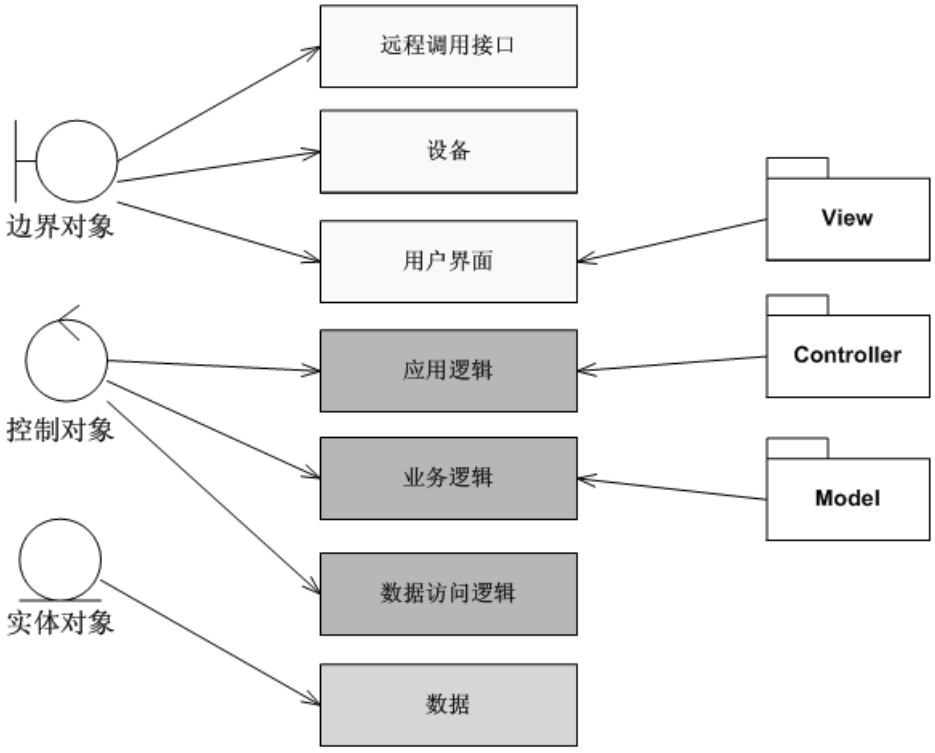


图 8-3　鲁棒图 3 元素和 MVC 的相似与不同

2.2 鲁棒图一例

图 8-4 所示，是银行储蓄系统的“销户”功能的鲁棒图。

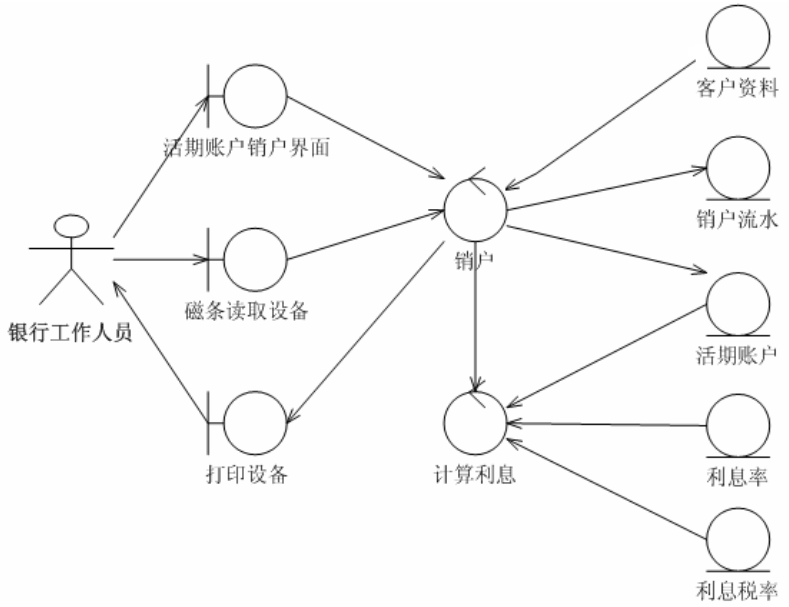


图 8-4　“销户”的鲁棒图

为了实现销户，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“活期账户”和“销户流水”信息。

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。

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依然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，应能够以默认值方式呈现，从而将程序崩溃的危险减为渲染不正常的危险。

而“鲁棒图（Robustness Diagram）”的作用有二，除了初步设计之外，就是检查用例规约是否正确和完善了。“鲁棒图”正是因为第二点作用，而得其名的——所以“鲁棒图（Robustness Diagram）”严格来讲所指不是“鲁棒性（Robustness）”。

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

> 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。我们的经验表明，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，将在绘制时序图时节省 3-4 倍的时间。
- 利于发现对象，这很重要，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问题。另外鲁棒分析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。

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。大家只需记住两个公式：

- 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》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的术语之一，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问题”是无可挑剔的。
>
> 但迄今为止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到“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。

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”。

第 3 节　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、注意事项这 4 个方面（如图 8-5 所示），帮助一线架构师快速提升初步设计的能力。

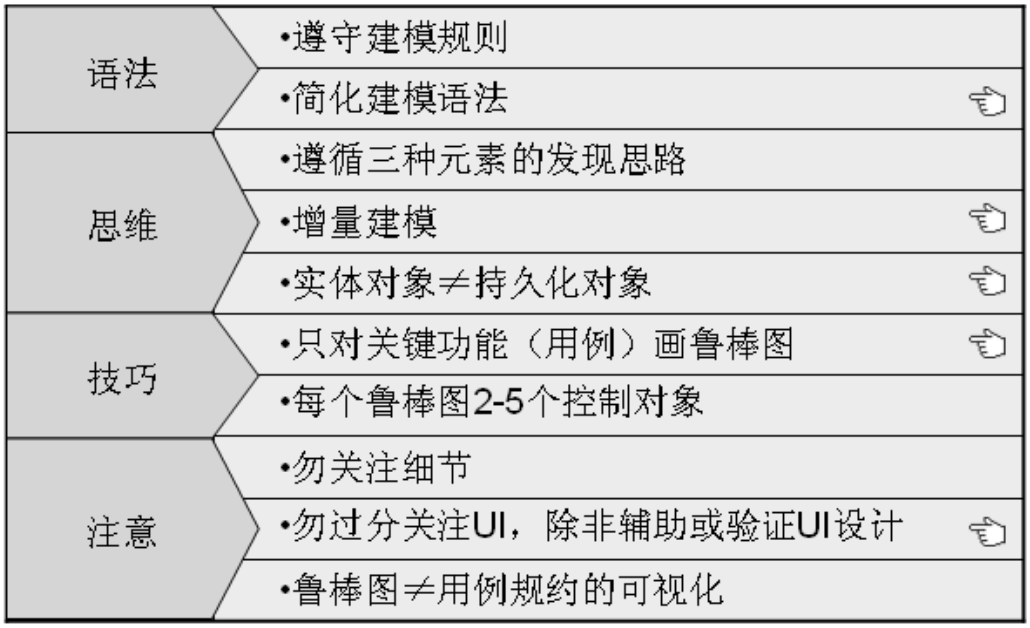


图 8-5　鲁棒图建模的 10 条经验

下面将逐一讲解鲁棒图建模的 10 条经验。

3.1 遵守建模规则

图 8-6 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。

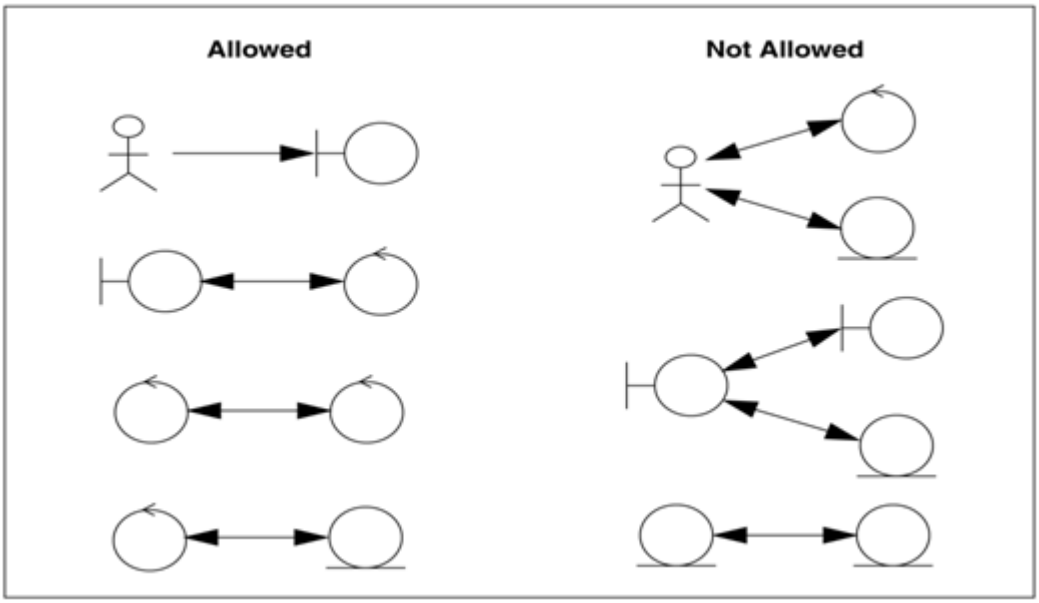


图 8-6　鲁棒图建模规则（图片来源：《UML 用例驱动对象建模》）

3.2 简化建模语法

图 8-7 展示了 ADMEMS 方法推荐的鲁棒图建模的语法。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。

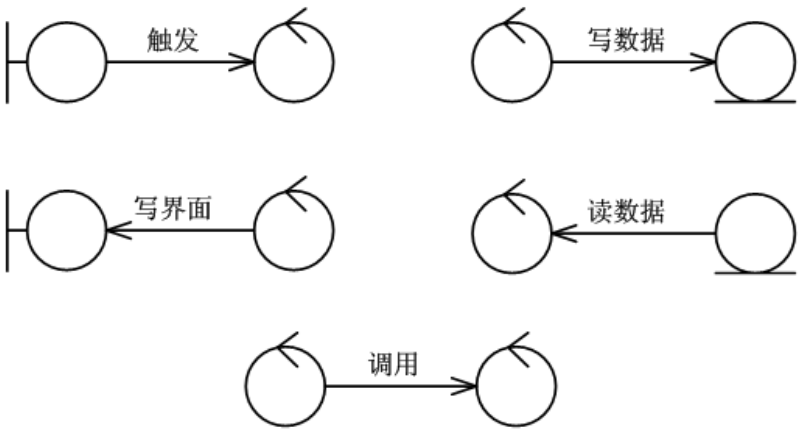


图 8-7　鲁棒图的语法

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。

3.3 遵循三种元素的发现思路

图 8-8 说明了发现鲁棒图 3 种元素的思维方式。

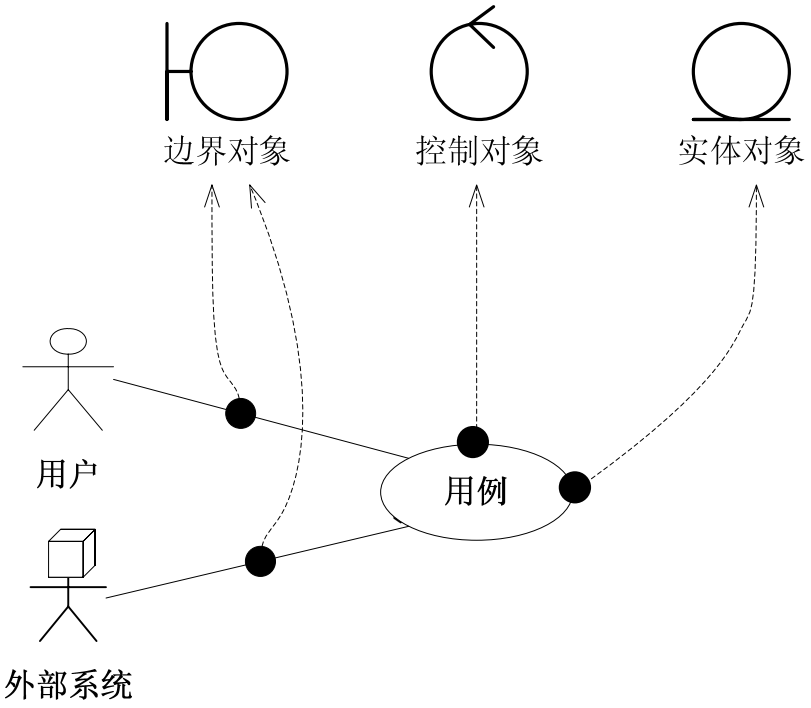


图 8-8　发现鲁棒图 3 种元素的思维方式

用例（Use Case）＝N 个场景（Scenario）。每个场景的实现都是由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。

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。

下面演示一种叫“增量建模”的技巧。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 8-9 所示，“你”认为压缩就是把“原文件”变成“压缩包”的处理过程，于是识别出了三个职责：

- 原文件
- 压缩包
- 压缩器（负责压缩处理）

图 8-9　增量建模：先识别最“明显”的职责

接下来，开始考虑职责间关系，并发现新职责。“压缩器”读“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。哦，还有“字典”……如图 8-10 所示。

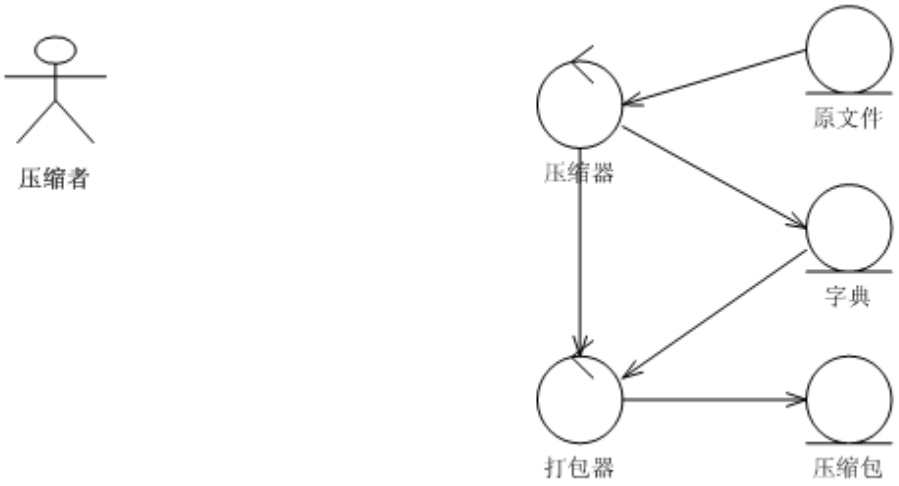


图 8-10　增量建模：开始考虑职责间关系，并发现新职责

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 8-11 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、

分卷压缩或是其他。

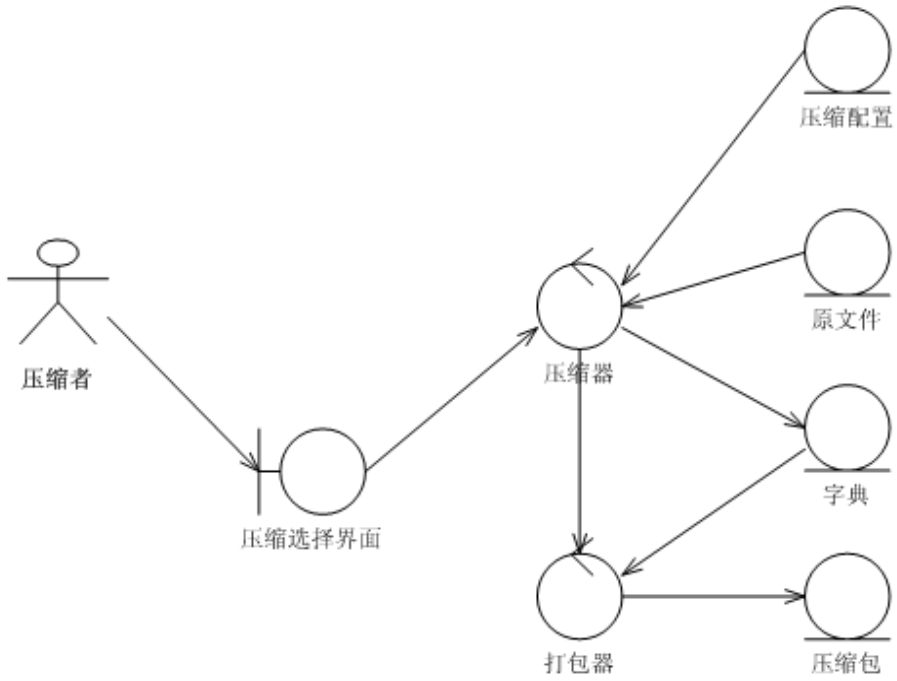


图 8-11　增量建模：继续考虑职责间关系，并发现新职责

……最终的鲁棒图如图 8-12 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。

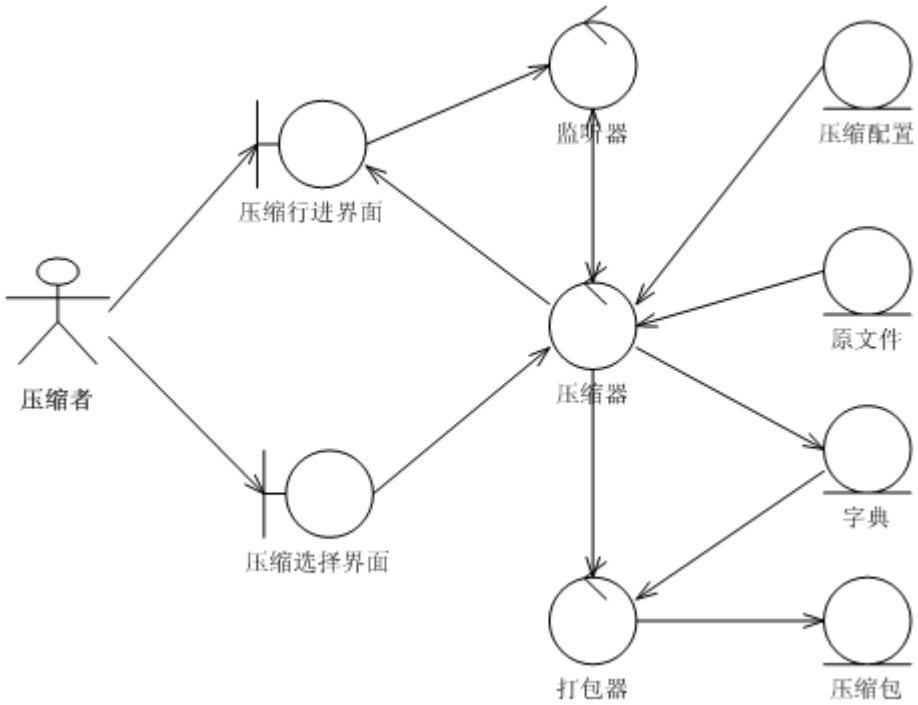


图 8-12　增量建模：直到模型比较完善

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”为我所用、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！

3.5 实体对象≠持久化对象

有的书上明确说“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。

实践中，有些系统需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。

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，在设计架构时不必每个功能都画鲁棒图。

3.7 每个鲁棒图 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，则是明显的“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 8-13 所示的鲁棒图，几乎没有任何意义。

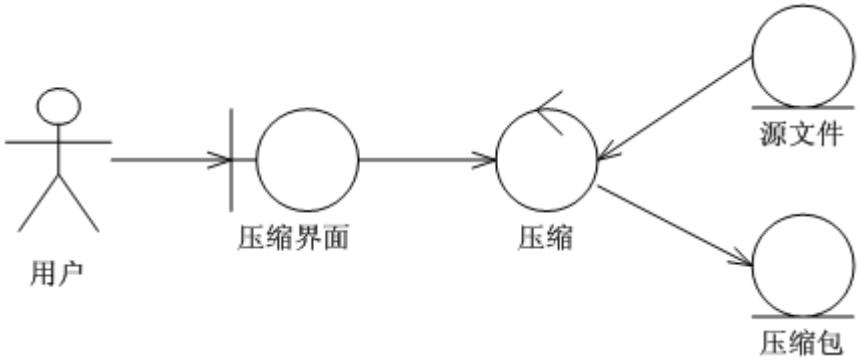


图 8-13　只有一个控制对象的鲁棒图明显“设计不足”

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：

- 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此细节问题。
- “客户资料”等实体对象，需要持久化吗？不关心，更不关心用 Table 还是用 File 或其他方式持久化。
- 而且，也没有标识控制流的严格顺序。

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，初步设计就没法做了。

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

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 8-14）。所以，二者有本质区别。

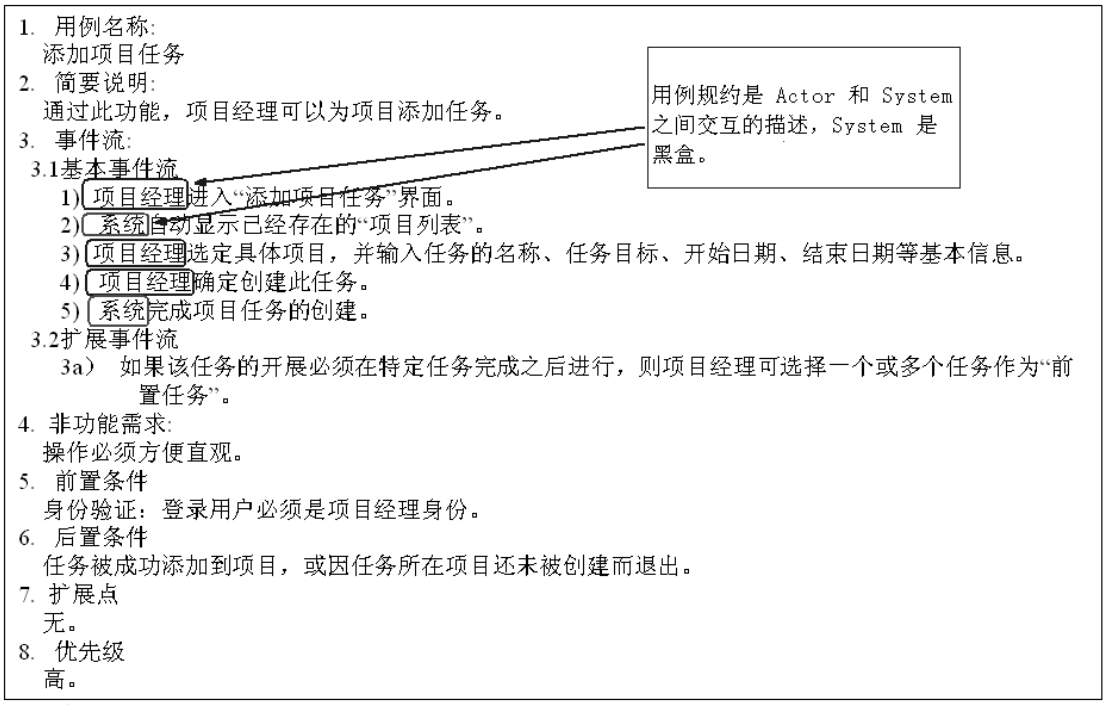


图 8-14　一个用例规约示例：需要意味着 System 保持黑盒

第 4 节　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“处方检查结果”相关的几个职责。

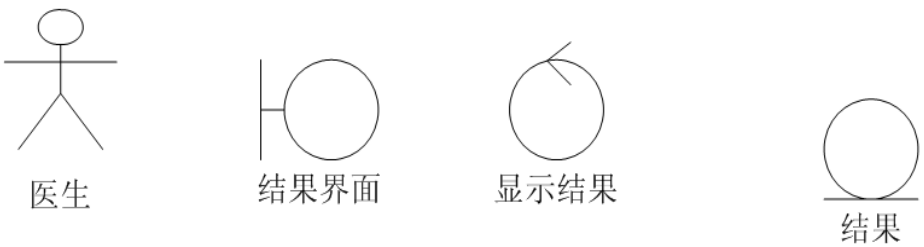


图 8-15　实时检查处方：先识别最“明显”的职责

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”信息。如图 8-16 所示。

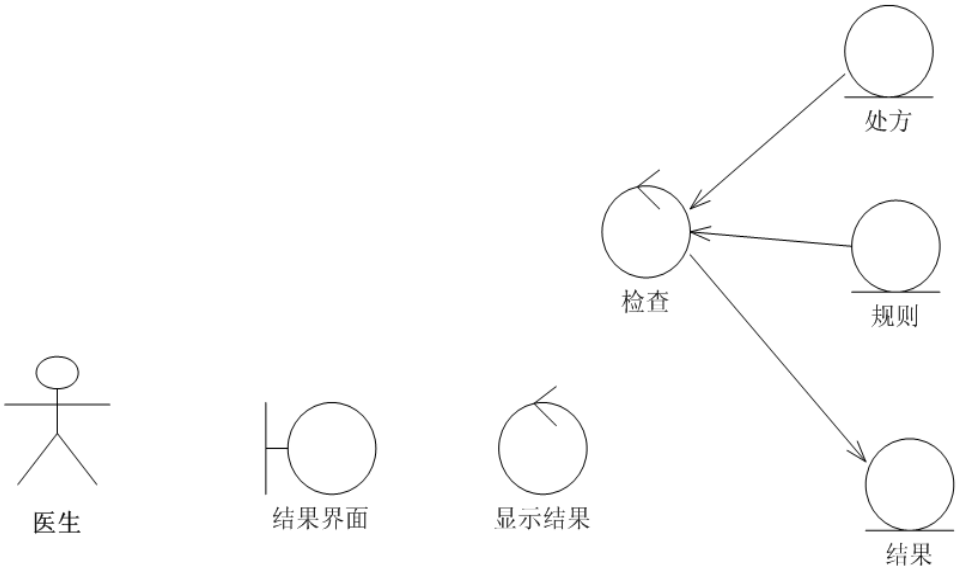


图 8-16　实时检查处方：开始考虑职责间关系，并发现新职责

OK，如此一来，解决了“结果怎么来的”这个问题。如图 8-17 所示。

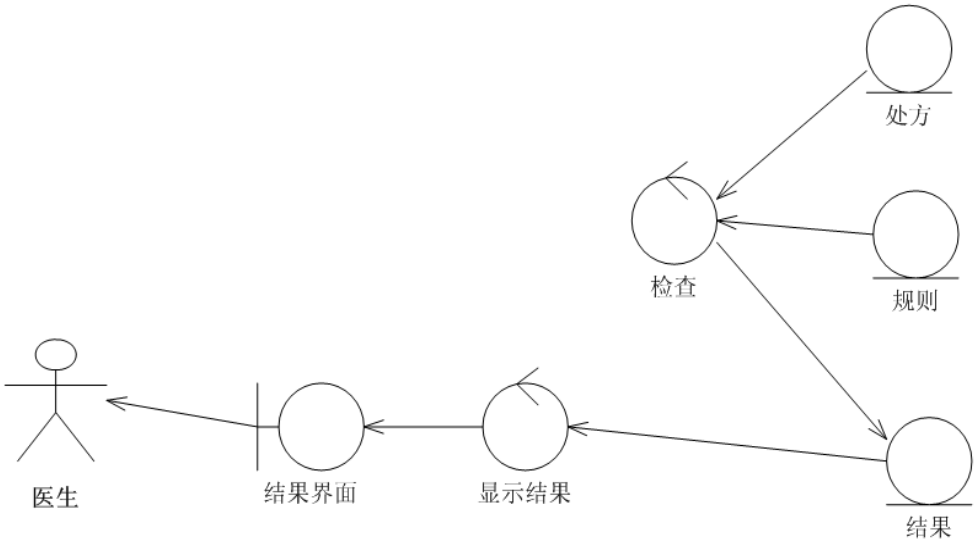


图 8-17　实时检查处方：开始考虑职责间关系，并发现新职责

继续同样的思维方式。如图 8-18 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。

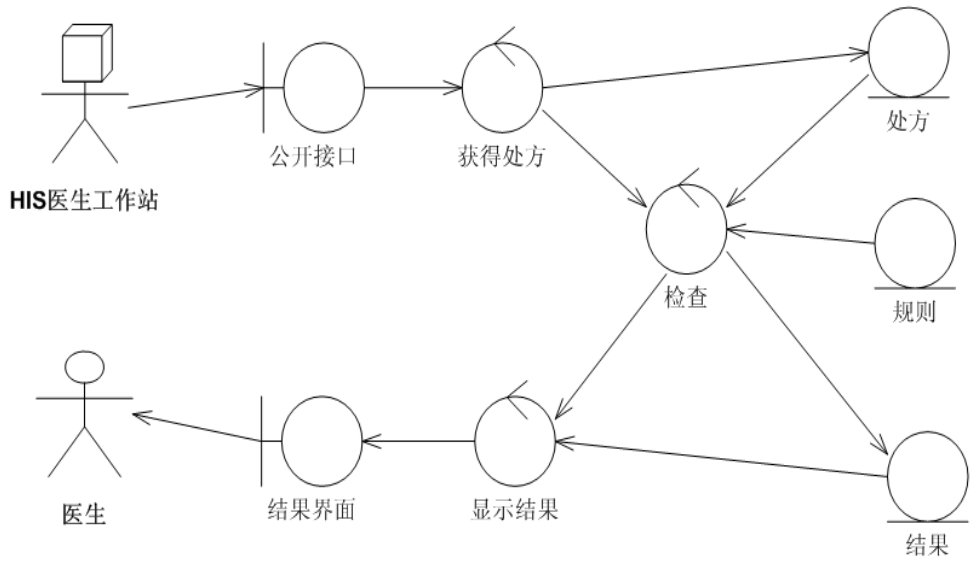


图 8-18　实时检查处方：继续考虑职责间关系，并发现新职责

……实时检查处方最终的鲁棒图如图 8-19 所示。它进一步考虑了“记录违规用药”这一具体功能场景的支持。

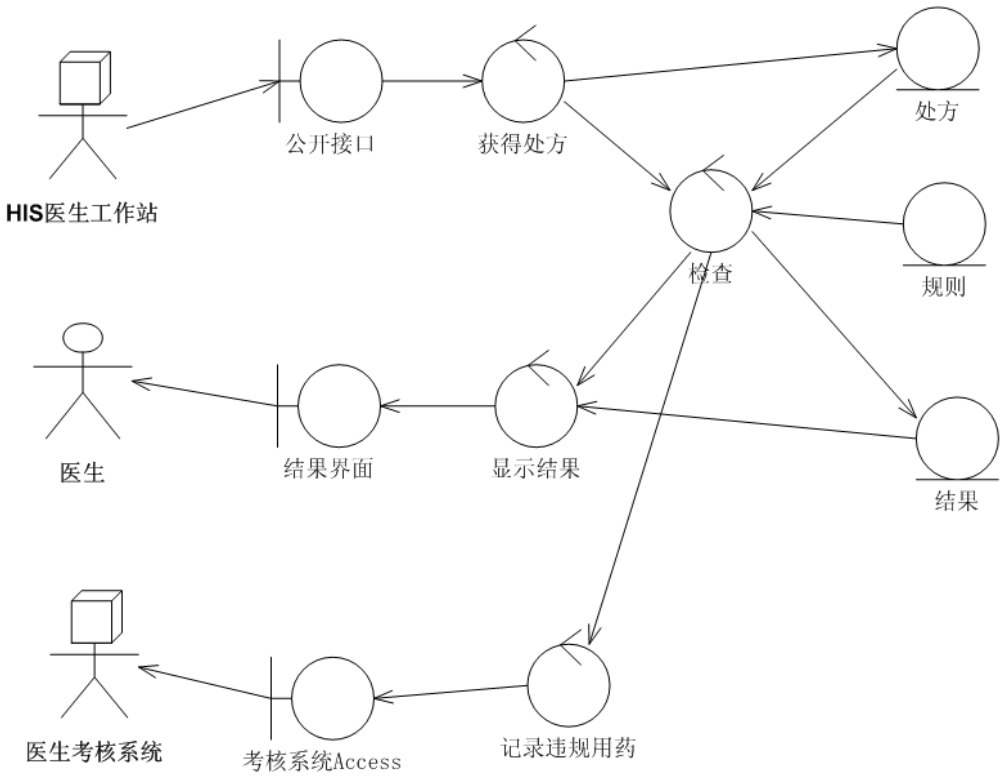


图 8-19　实时检查处方：直到模型比较完善

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 8-20 所示。

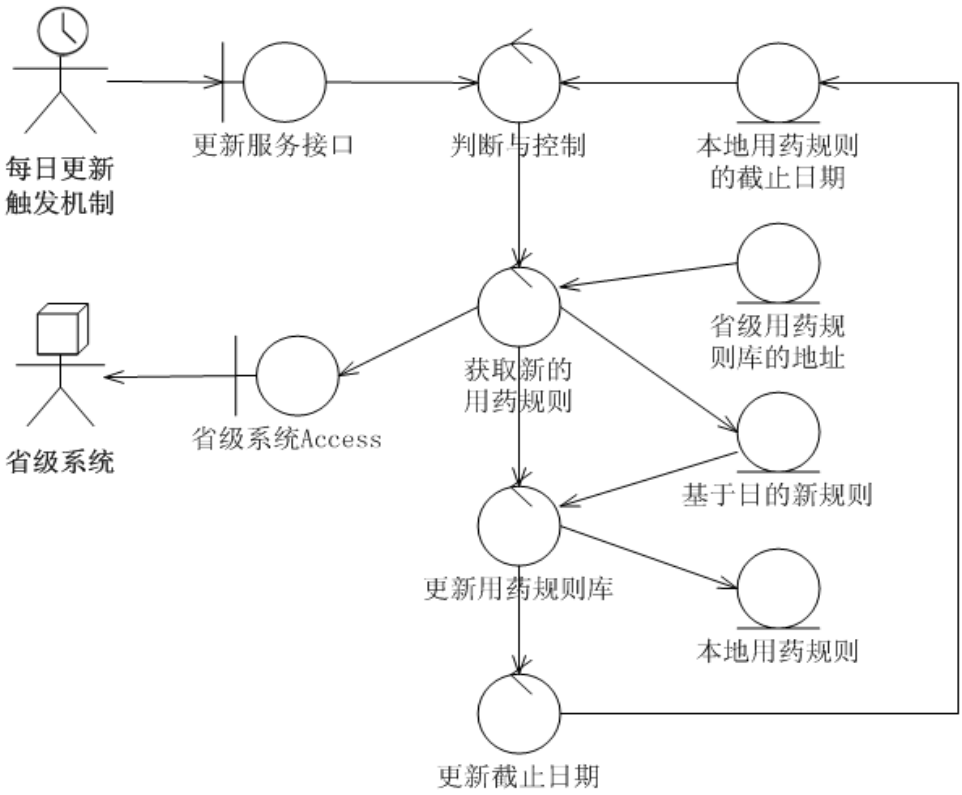


图 8-20　自动更新用药规则：基于鲁棒图的初步设计

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》

对软件设计者来说，被简单、直观地分割，并具有最小内部耦合的内部结构就是美的。

——Robert C. Martin，《软件之美》

划分子系统……，定义接口……，这些经典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，本章将告诉你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。

第 1 节　划分子系统的 3 种必用策略

本书已不止一次强调，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必做工作而言，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

1.1 分层（Layer）的细化

分层是最常用的架构模式，人们常这么说。而我的总结稍有不同：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。

于是不难理解，架构师在划分子系统时必然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 说明了基于三层架构进行“分层细化”的一种方式。

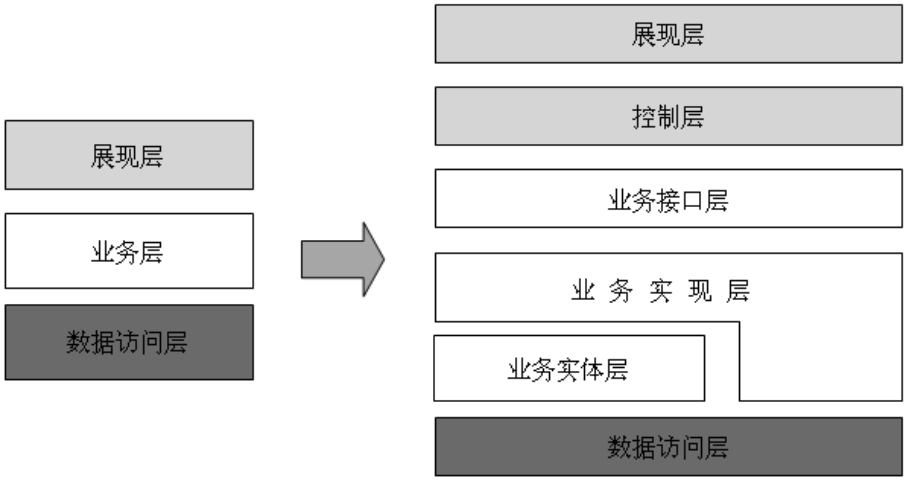


图 13-1　基于三层架构进行“分层细化”的一种方式

三层架构、或四层架构的“倩影”经常出现在投标时、或市场彩页中，于是有人戏称之为“市场架构”。的确，直接用三层架构或四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。

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，是一线程序员们经常遇到的烦恼。

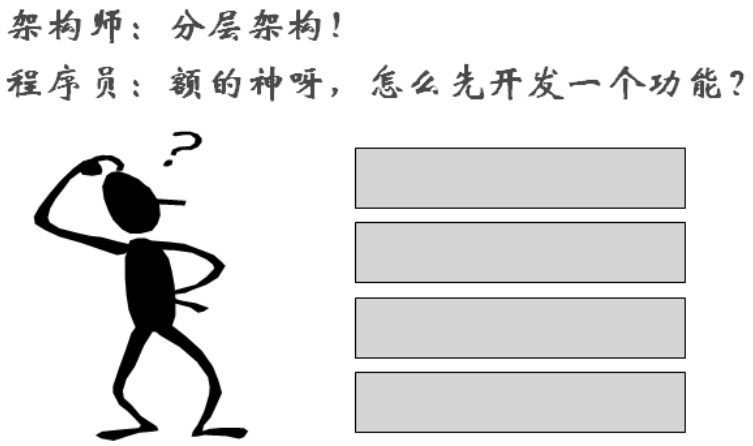


图 13-2　只有分层，如何迭代开发呢？

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（我说的是必须）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。

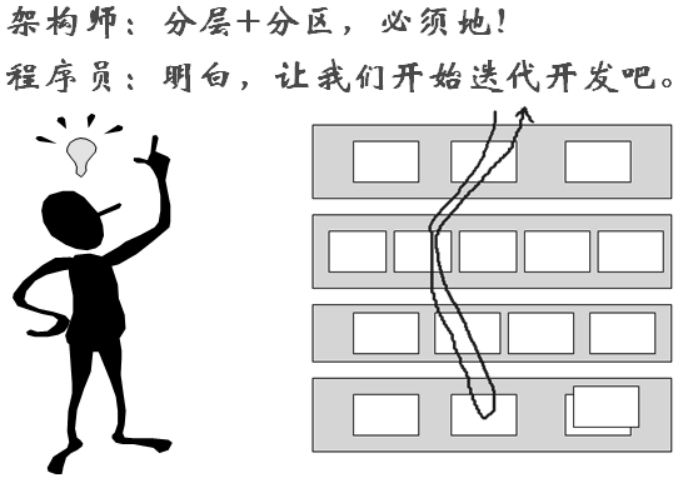


图 13-3　架构中引入分区（Partition），支持迭代开发

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。

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重要！那么，什么是机制呢？

本书为“机制”下的定义：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”的区别可以务实地概括为：

> 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。

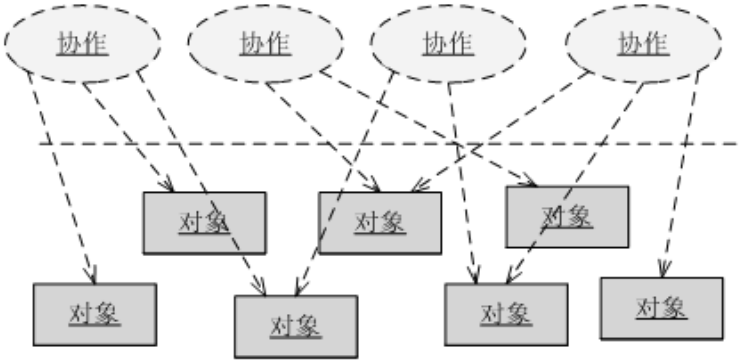


图 13-4　直接组装也叫协作

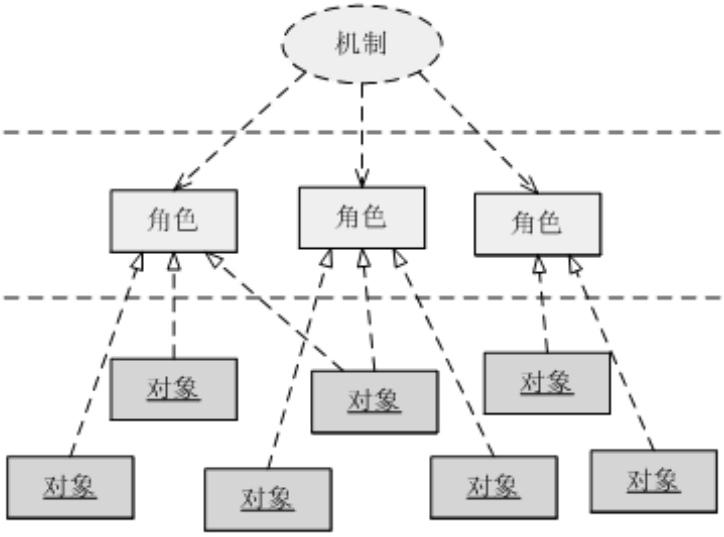


图 13-5　基于抽象角色的协作才叫机制

对于编程实现而言，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。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。

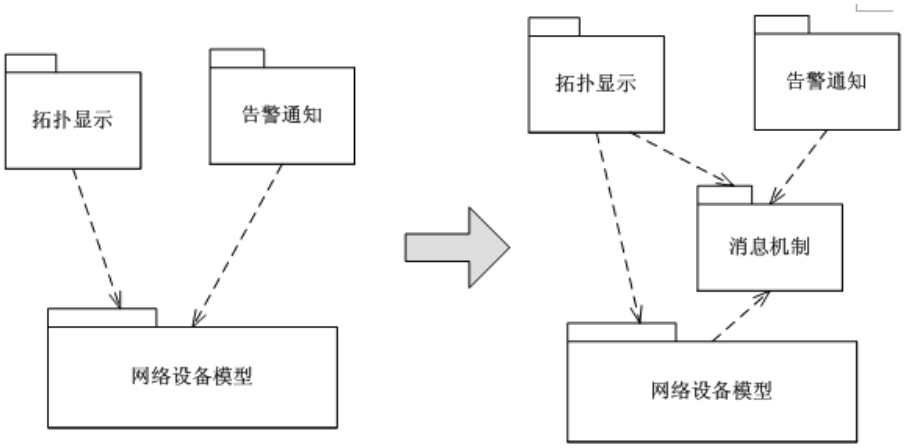


图 13-6　提取通用机制的例子

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。

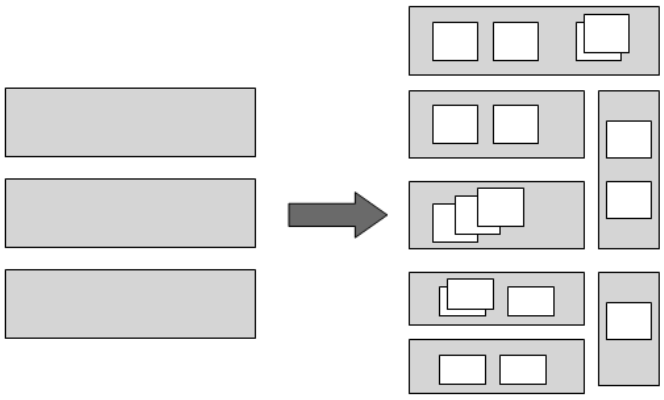


图 13-7　划分子系统必须三管齐下，综合运用 3 种手段

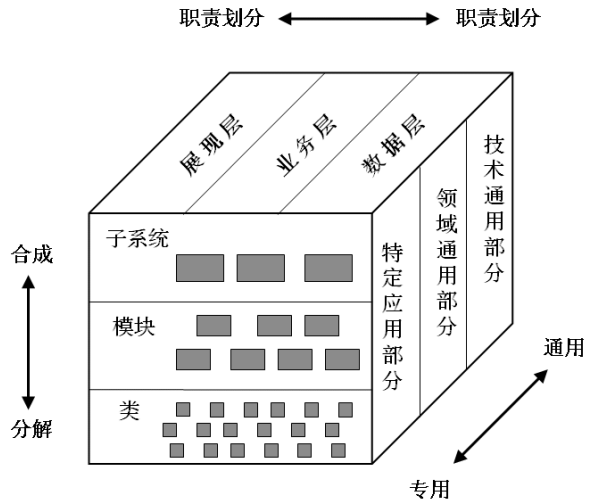
笔者曾在《软件架构设计》一书中指出：

> 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需构造新的“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。……



此图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、机制的提取这三种划分子系统的手段之间的关系：它们处在思维的三个维度上。

首先，分层和机制位于不同的维度：职责维、以及通用维（如图 13-8 所示）。

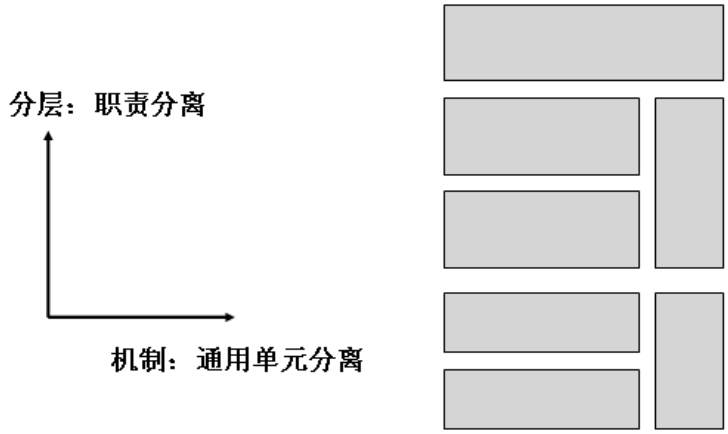


图 13-8　分层和机制位于不同的维度

另外，是否引入分区，设计所“覆盖”的 Scope 是完全相同的。原因何在？因为层的粒度较大，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 13-9 所示）。

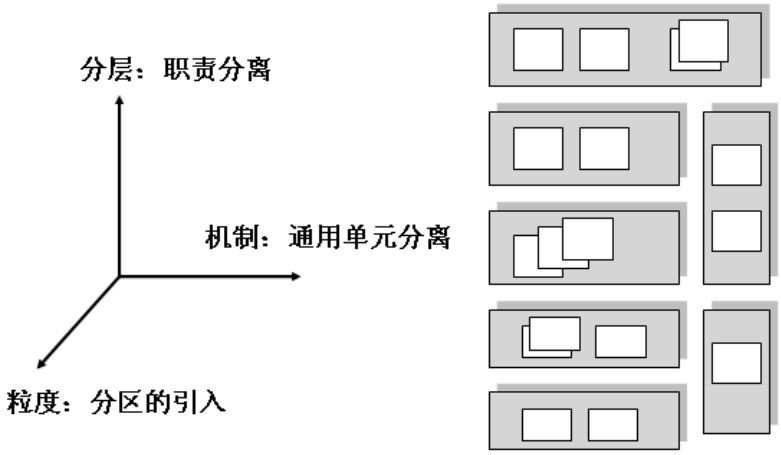


图 13-9　粒度维：分区的引入

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，架构师切记应三管齐下、综合运用。

1.5 探究：3 种子系统划分策略背后的 4 大原则

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：

- 原则 1：职责不同的单元划归不同子系统
- 原则 2：通用性不同的单元划归不同子系统
- 原则 3：需要不同开发技能的单元划归不同子系统

♦ 原则 4：兼顾工作量的相对均衡，把太大的子系统进一步做切分

如图 13-10 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。

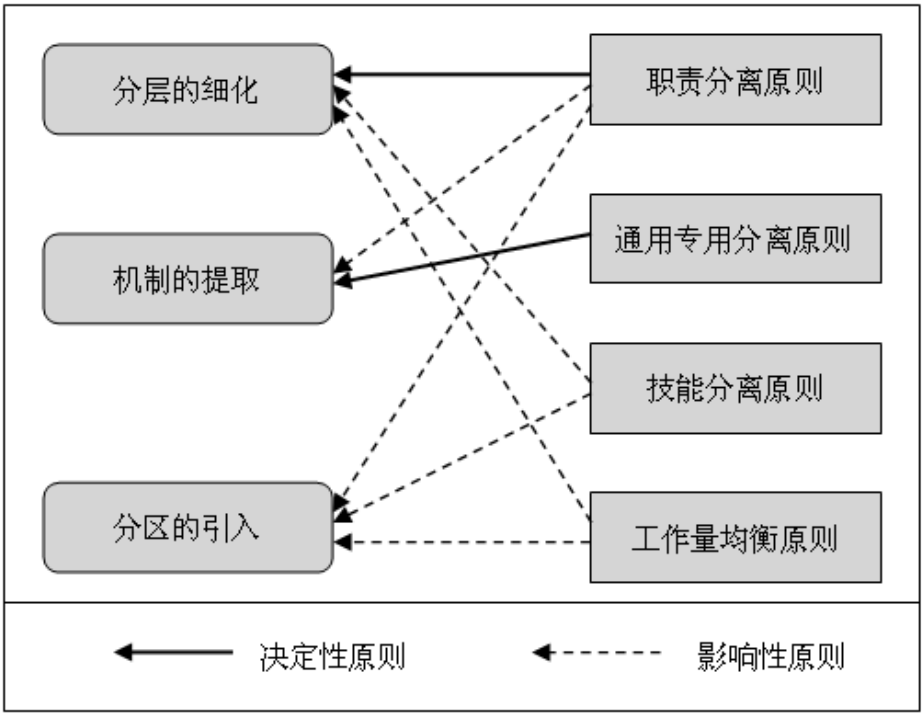


图 13-10　3 种子系统划分策略背后的 4 大原则

第 2 节　接口设计的事实与谬误

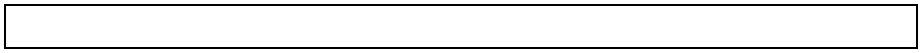
世界是复杂的，很多东西难以直接获得。例如，直接追求幸福，是永远追不到的。（《乐在工作》一书中说幸福是副产品。）

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，虽然“自治”是正确的，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 发表了第一篇关于层次结构的论文。1972 年，Parnas 发表论文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。

之所以要将软件模块化，是为了解决复杂性更高的大问题；而对问题进行分解、分别解决小问题，则只是手段。每个架构师必须牢记：



“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下进行考虑。作为架构师，你想要设计接口，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 13-11 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。

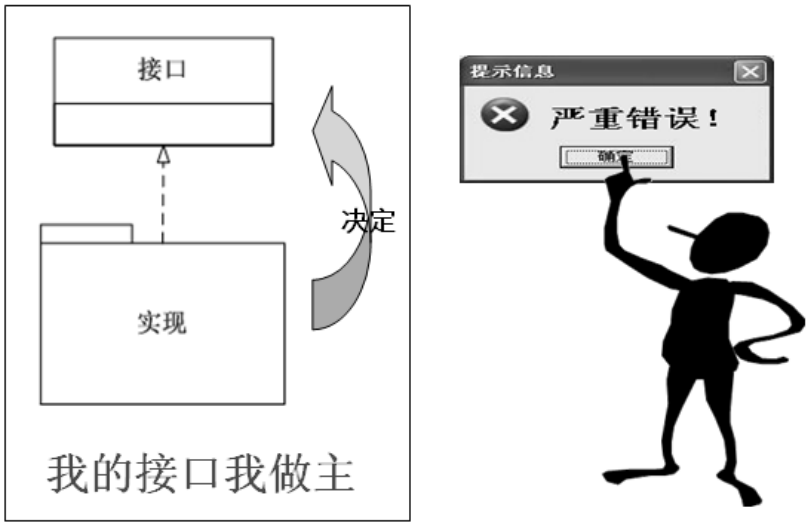


图 13-11　协作决定接口，“我的接口我做主”的观点是错误的

“世界上最短的距离不是直线”，又多了一个例证。

第 3 节　逻辑架构设计的整体思维套路

3.1 整体思路：质疑驱动的逻辑架构设计

要点有二：

- 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——“缓释”的控制者必然是人，是架构师。

请看图 13-12 所示的“药力释放机理”——逻辑架构设计的整体思维套路。

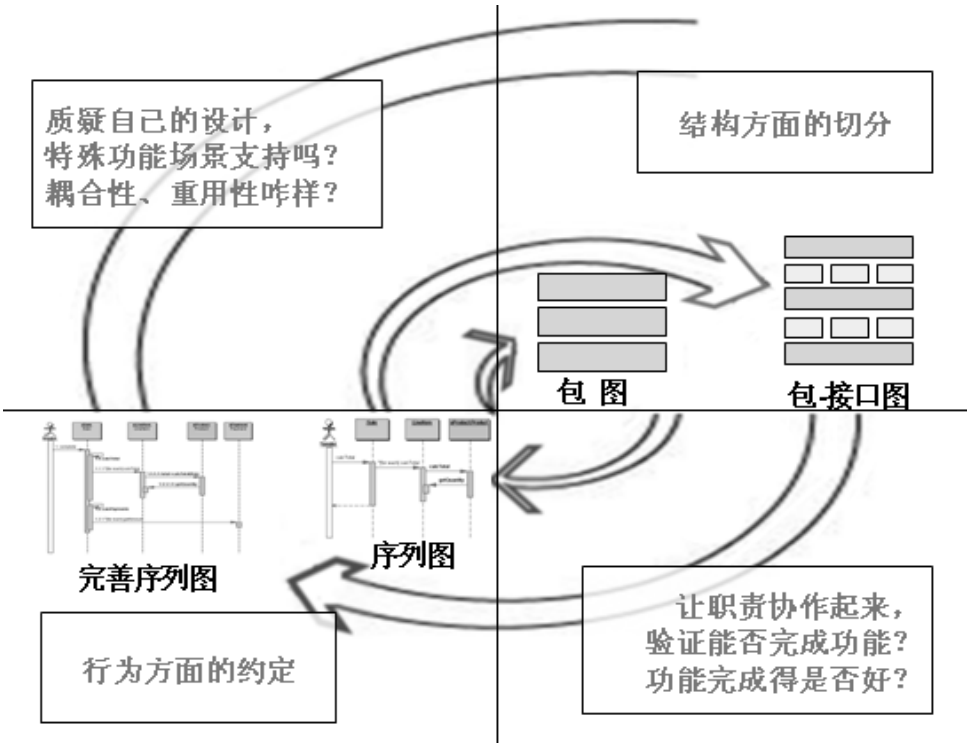


图 13-12　逻辑架构设计的整体思维套路

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。

3.2 过程串联：给初学者

第 1 步，根据当前理解切分。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。如图 3-13 所示。

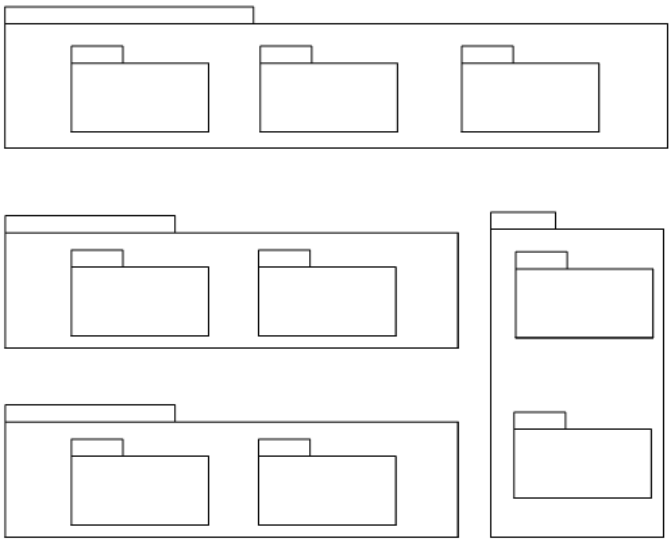


图 3-13　　第 1 步：根据当前理解切分

第 2 步，找到某功能的参与单元。若找不到，或明显却单元，就可以直接返回第 1 步了，以补充遗漏的职责单元。如图 3-14 所示。

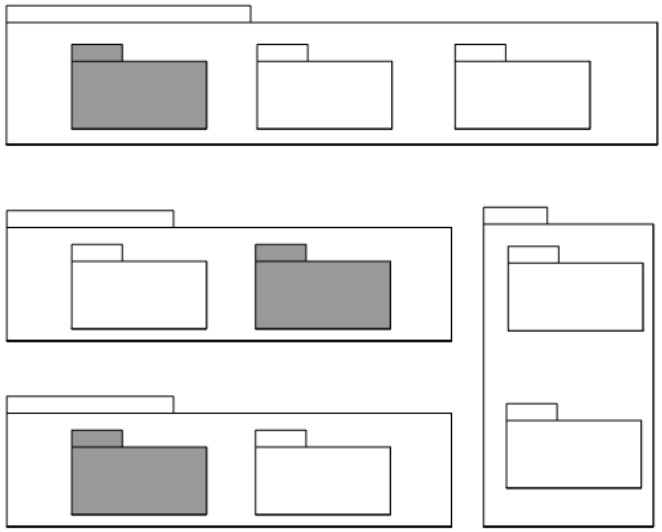


图 3-14　　第 2 步：找到某功能的参与单元

第 3 步，让它们协作完成功能。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、完成的怎么样？

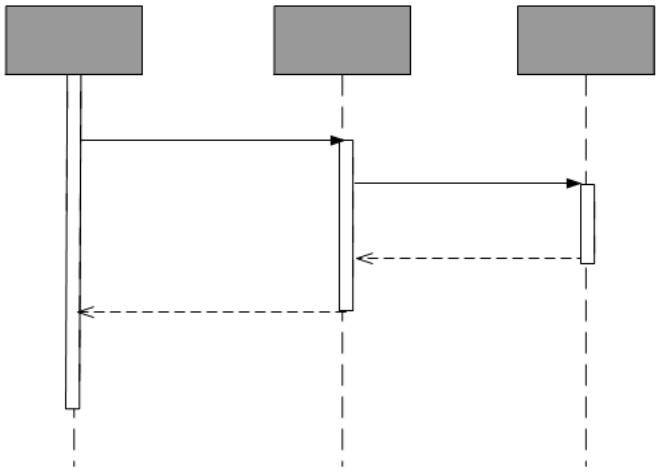


图 3-15　第 3 步：让它们协作完成功能

第 4 步，质疑并推进设计的深入。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。

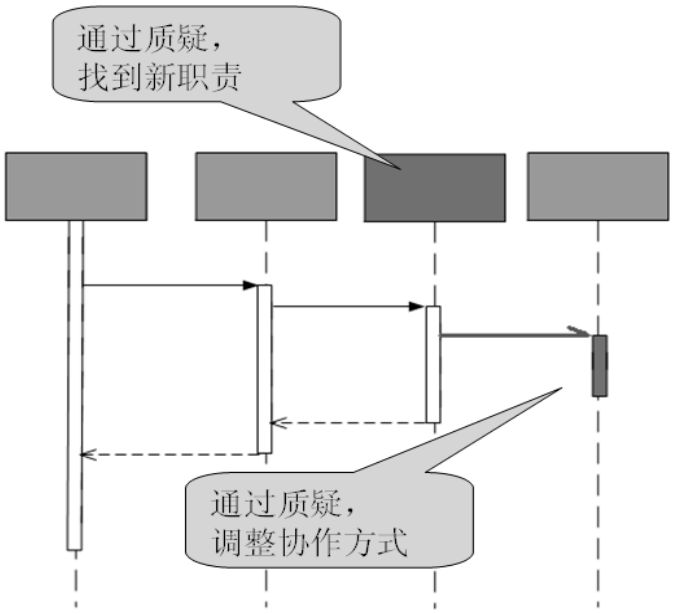


图 3-16　第 4 步：质疑并推进设计的深入

3.3 案例示范：自己设计 MyZip

图 3-17 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。

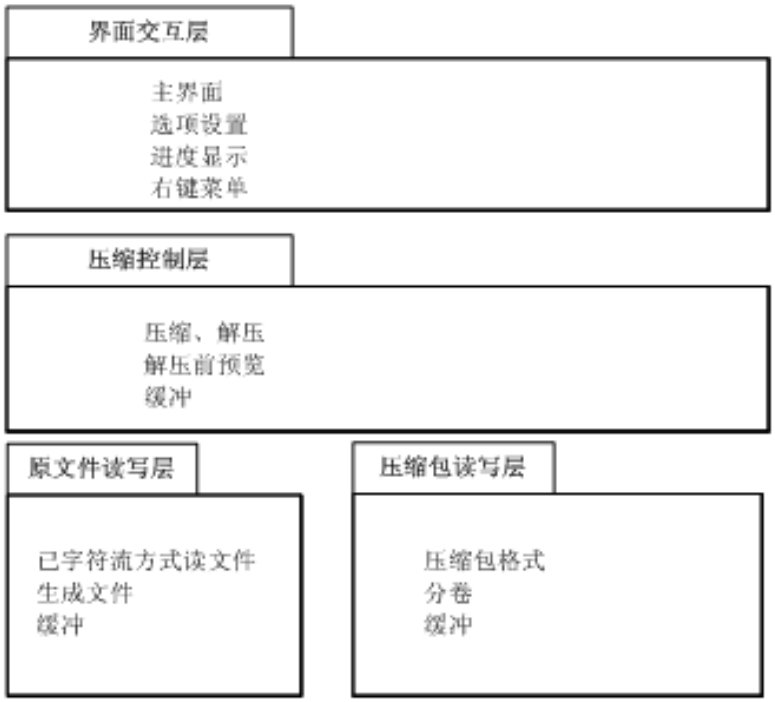


图 3-17　MyZip 的概念架构设计

下面，主要演示如何按照质疑驱动的思路，设计 MyZip 的逻辑架构视图。

首先，考虑结构方面的切分，如图 3-18 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，二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，从而简化了使用缓冲区的接口。

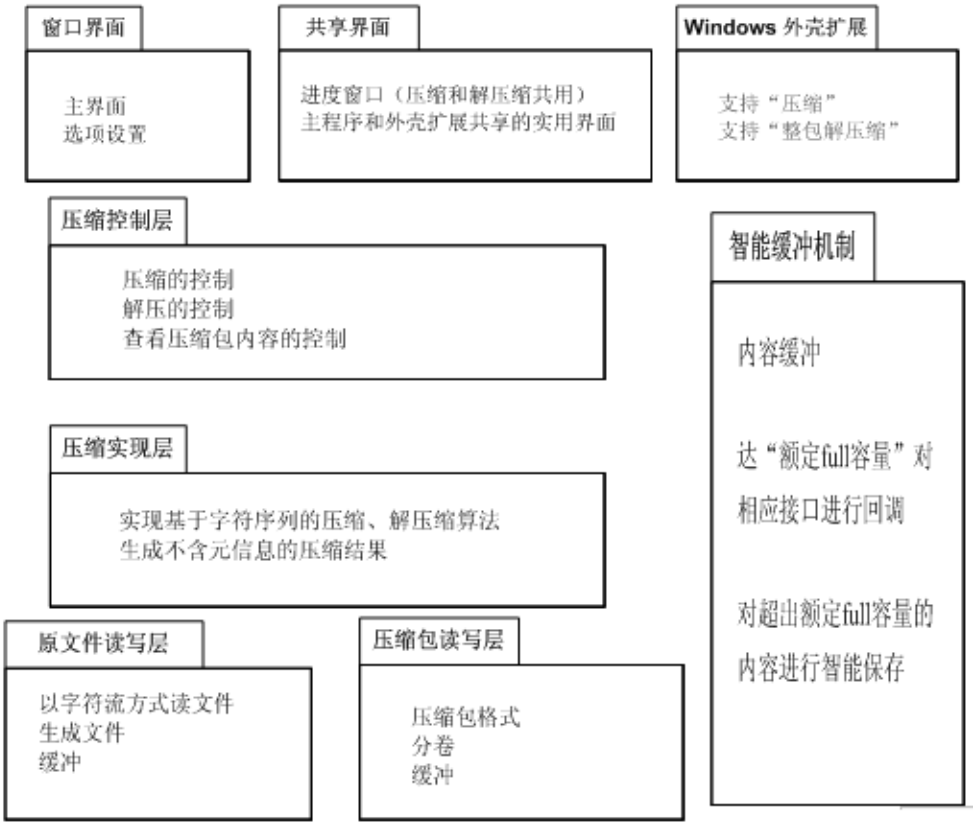


图 3-18　MyZip 逻辑架构设计：首先考虑结构方面的切分

然后，让切分出的职责协作起来，验证能否完成功能。图 3-19 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，回忆本书提倡的“增量建模”技巧——不要急于“一口吃个胖子”。

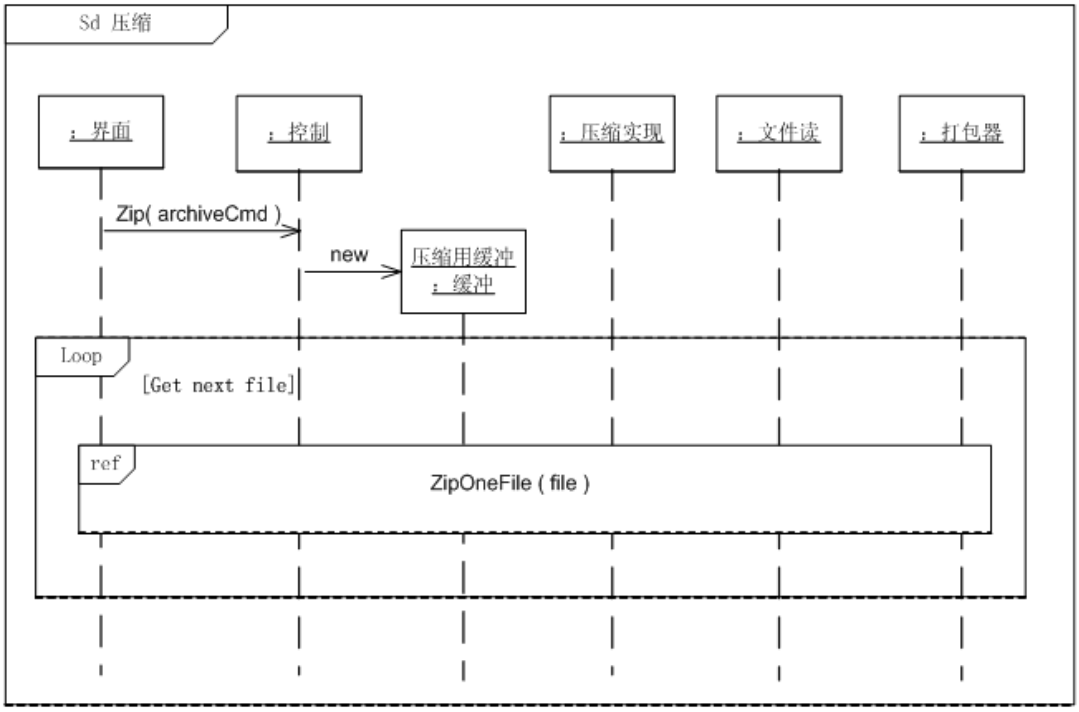


图 3-19　MyZip 逻辑架构设计：协作以验证能否完成“压缩”功能

如此循环，早晚要定义子系统的接口。图 3-20 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）。

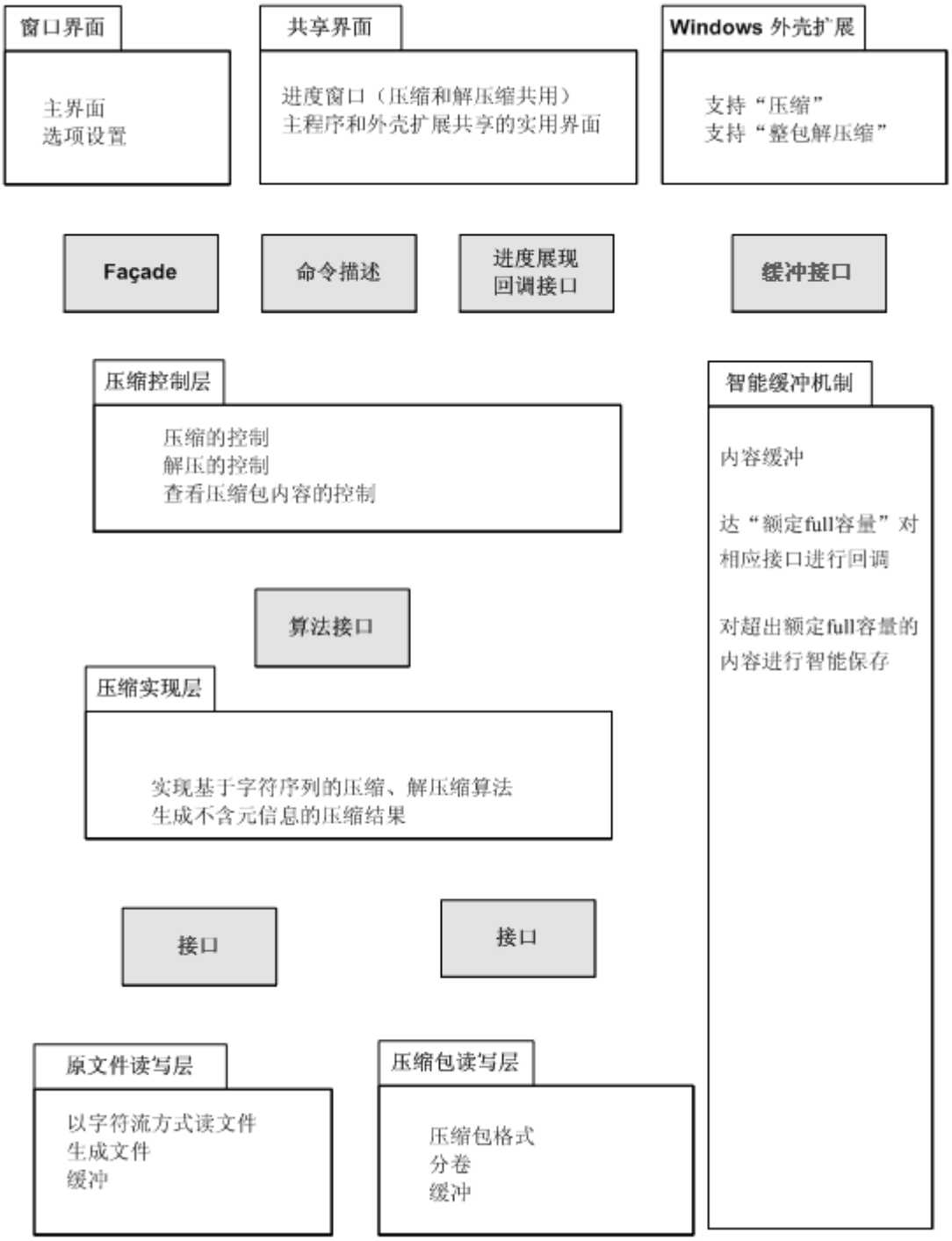


图 3-20　MyZip 逻辑架构设计：包-接口图

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 3-21 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”不需感知 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）。

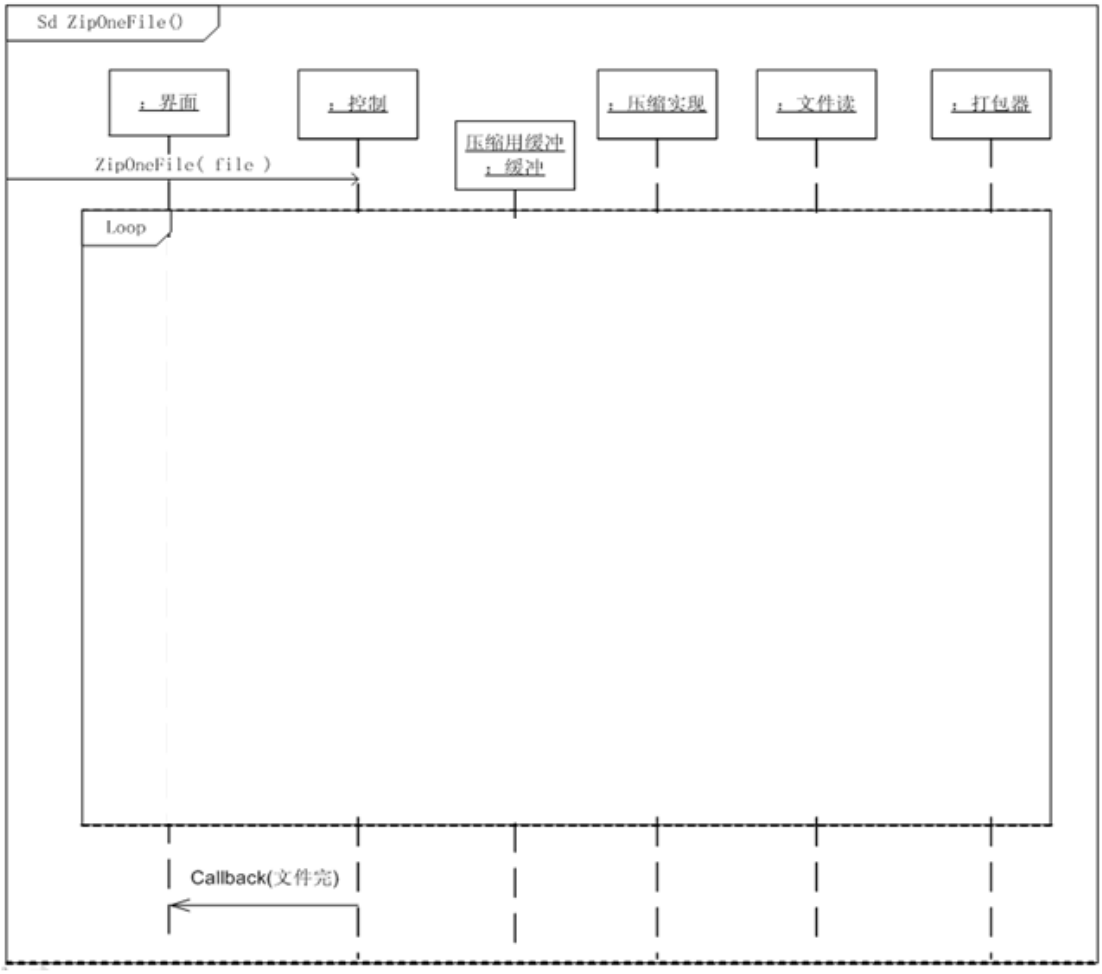


图 3-21　　MyZip 逻辑架构设计：每个文件相关的压缩处理

到了图 3-22 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……

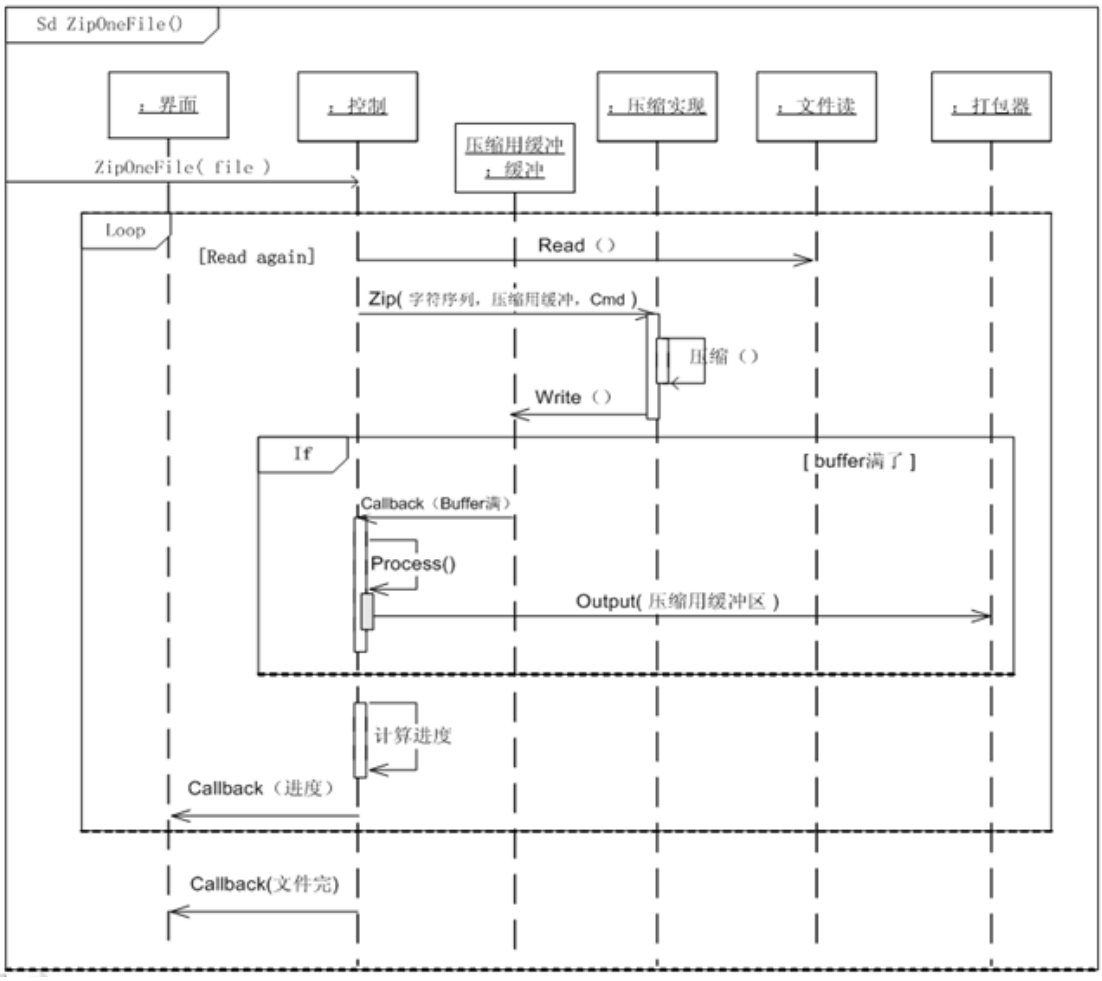


图 3-22　MyZip 逻辑架构设计：每个文件相关的压缩处理（续）

第 4 节　更多经验总结

4.1 逻辑架构设计的 10 条经验要点

图 3-23 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是刚刚将过的，其他几点仅在后续小节简述之。

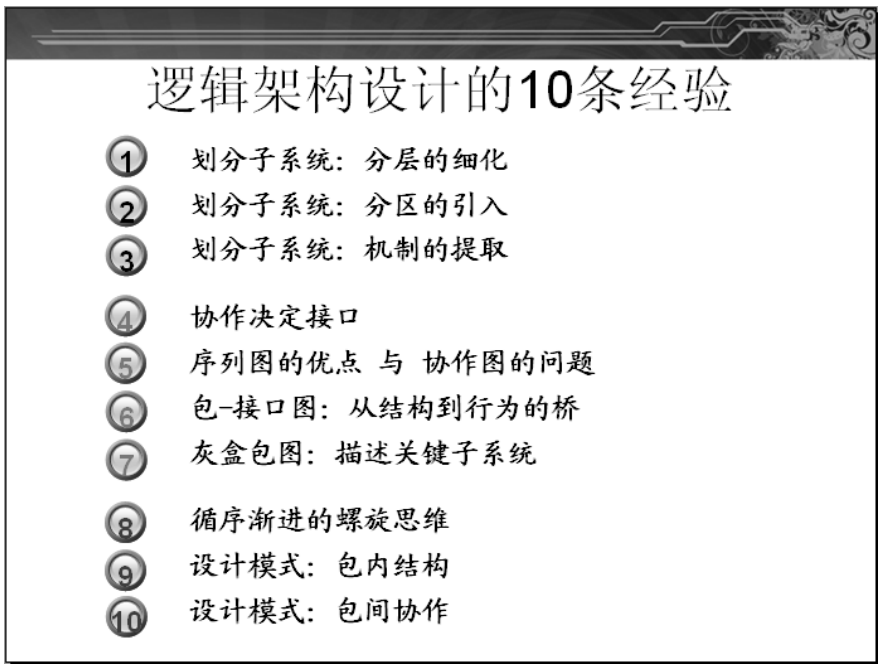


图 3-23　逻辑架构设计的 10 条经验要点

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 2 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 3-24 说明了灰盒包图的意义，它打破了“子系统黑盒”，它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。

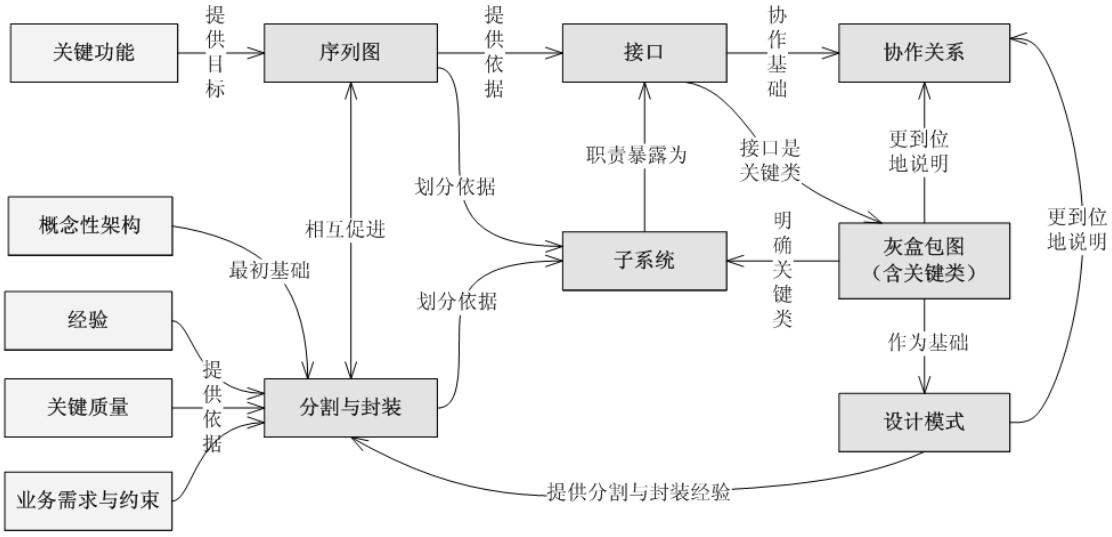


图 3-24　灰盒包图在逻辑架构思维中的“位置”

例如，来对比图 3-25 和 3-26——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。

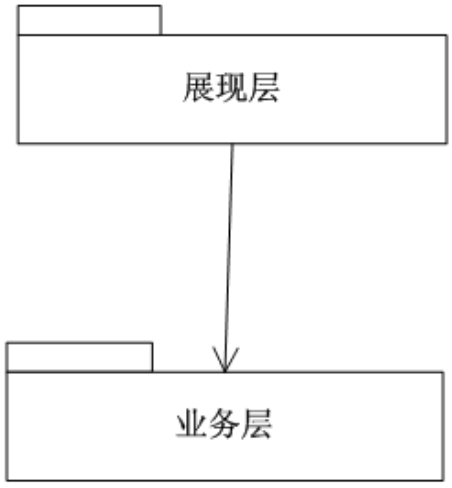


图 3-25　黑盒包图：设计不足

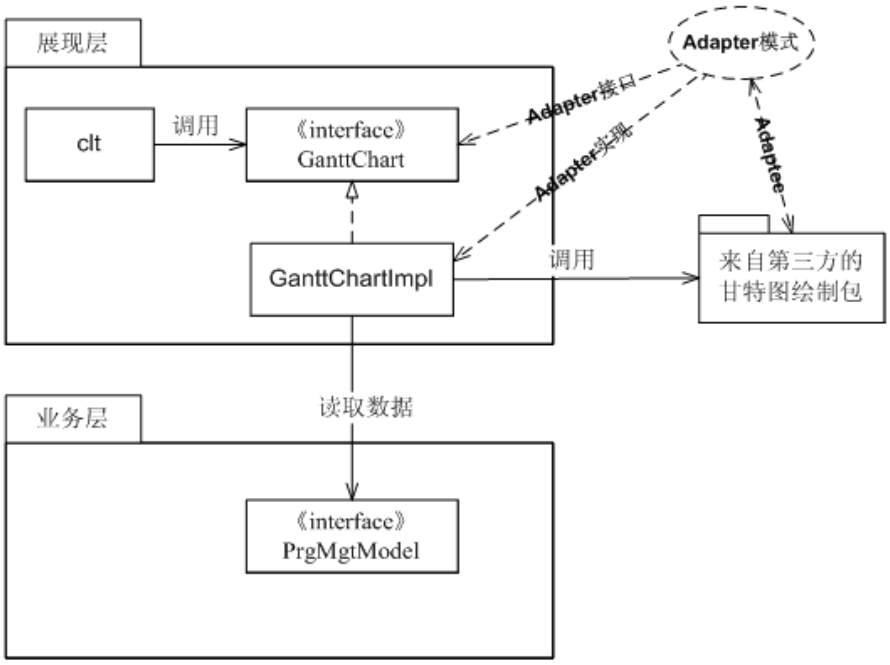


图 3-26　灰盒包图：聪明的折衷

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 3-20）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，应结构设计和行为设计相分离。因为这样，才利于更有效地思维。不信？请看图 3-27 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。

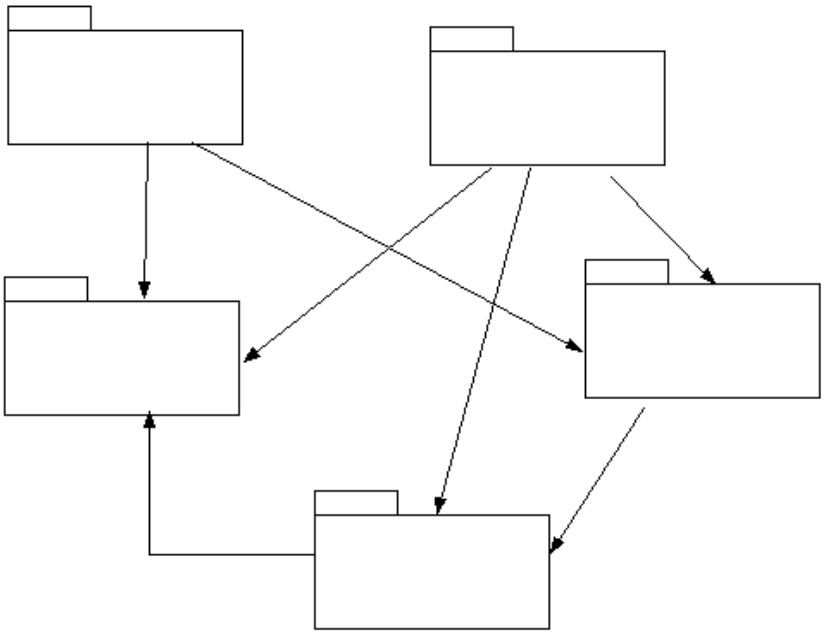


图 3-27　思维混乱的原因：将结构和行为过多地混在了一起

于是不难理解，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 UML 1.4 中它和序列图等价，但从形式上它的“结构气”太重）。

第 5 节　贯穿案例

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》

很多架构师对数据分布问题非常困惑——毫无章法、没有大局观。

的确，确定数据分布方案是数据架构设计的难点。而且越是大系统，数据分布越为关键。

本章结合案例，讲解如何运用数据分布的 6 种具体策略。

第 1 节　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）

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”。

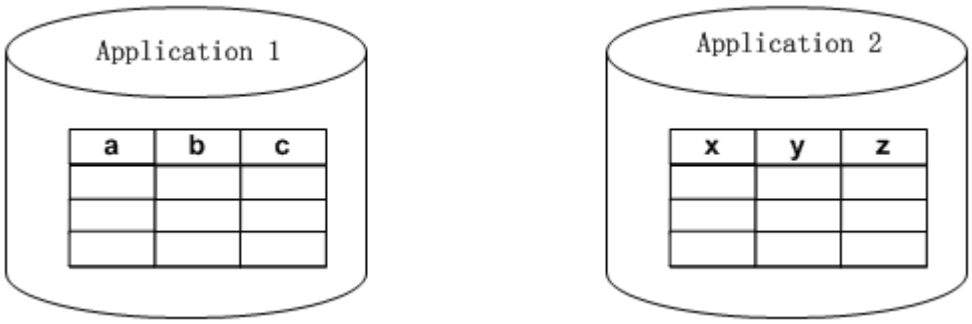


图 15-1　数据分布策略“独立 Schema”示意图

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。

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。

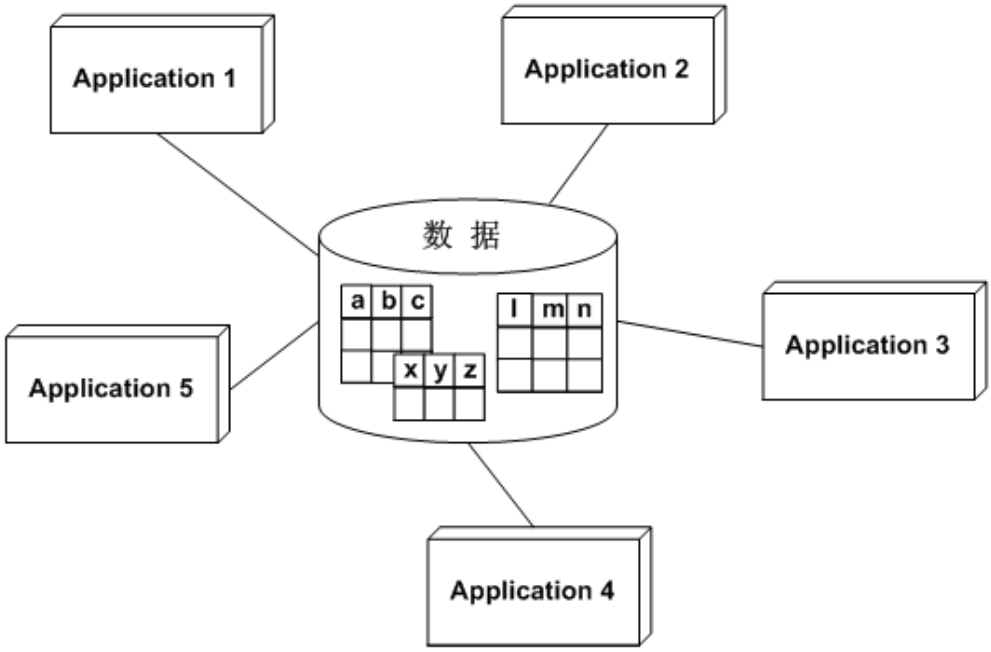


图 15-2　数据分布策略“集中”示意图

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。

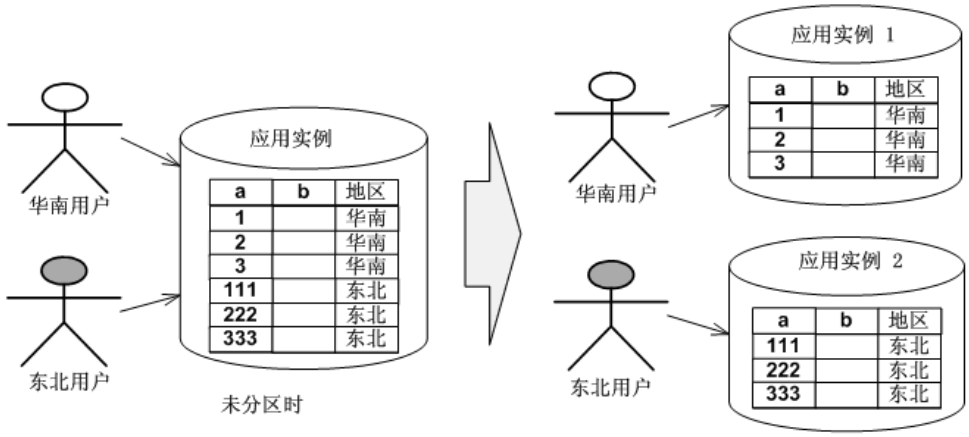


图 15-3　“水平分区”示意图

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。

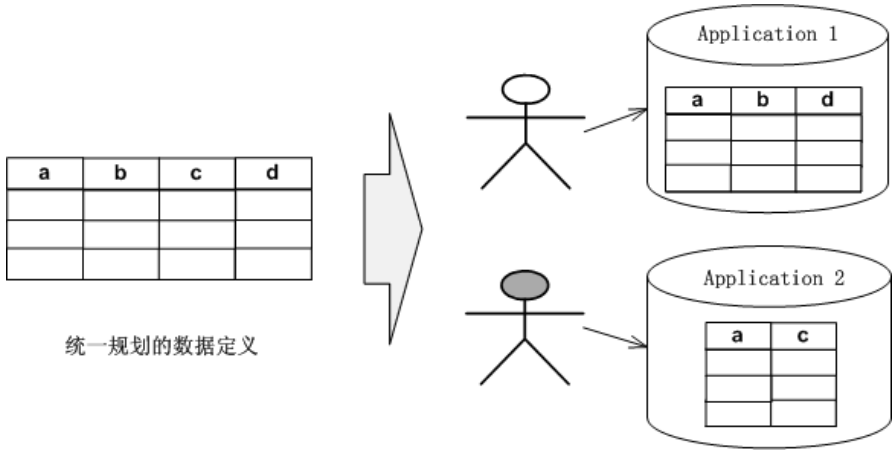


图 15-4　“垂直分区”示意图

另外，需要特别说明的是，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。

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，叫做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

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，利于进行有针对性地优化（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，便于加强可管理性和安全控制等

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

1.6 重组（Reorganized）

业务决定功能，功能决定模型。每当我们遇到数据模型不同的情况，必然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，重组策略需要进行数据传递，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。

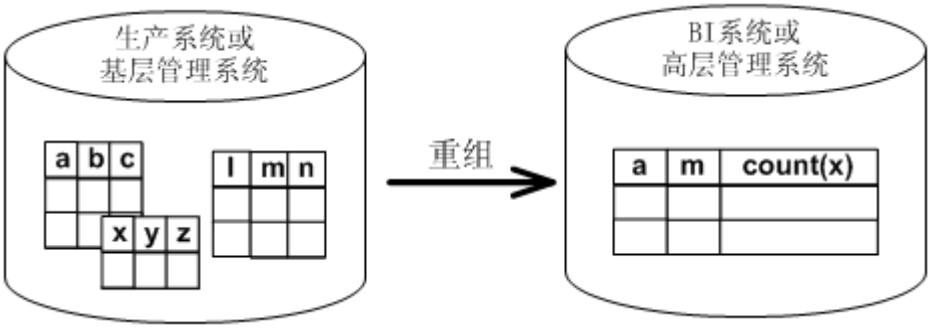


图 15-7　数据分布策略“重组”示意图

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，如果总公司只需要掌握各分公司的财务、生产等概况信息，那么就并不需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。

第 2 节　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、以及不同点。

2.1 6 种策略的二维比较图

一图胜千言，笔者再次用图揭示“复杂背后的简单”。

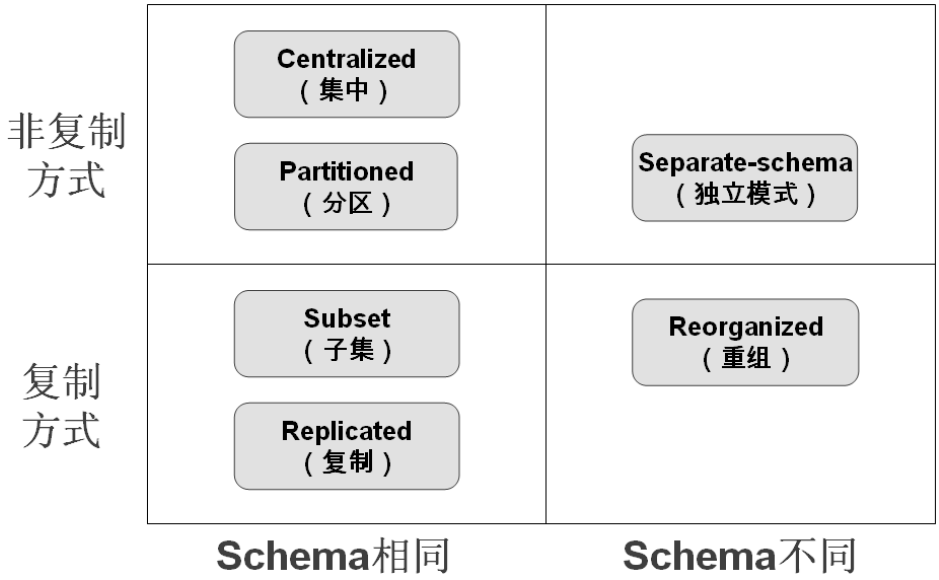


图 15-8　　6 种策略的二维比较图

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 2 种方式：非复制方式、复制方式。

非复制方式包括三种具体策略：集中、分区、独立 Schema。

复制方式也包括三种具体策略：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 2 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。

2.2 可靠性、可伸缩性、可管理性等的“冠军”

选择数据分布策略，还需要关注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（如图 15-9 所示）。

	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					

图 15-9　可靠性、可伸缩性、可管理性等的冠军

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，就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，这种观点对了一半——当数据的只读式“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，我前面讲：“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了。可管理性应该视原始问题的复杂程度而论，是相对的不是绝对的。

数据一致性冠军：集中。所有用户面对同样的数据，方便保持数据的一致性。

第 3 节　　数据分布策略的 3 条应用原则

读到这里，你已全面了解了数据分布的 6 种策略。下面，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）

3.1 合适原则

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。

3.2 案例：电子病历 vs. 身份验证

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。……于是，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 15-10 所示。

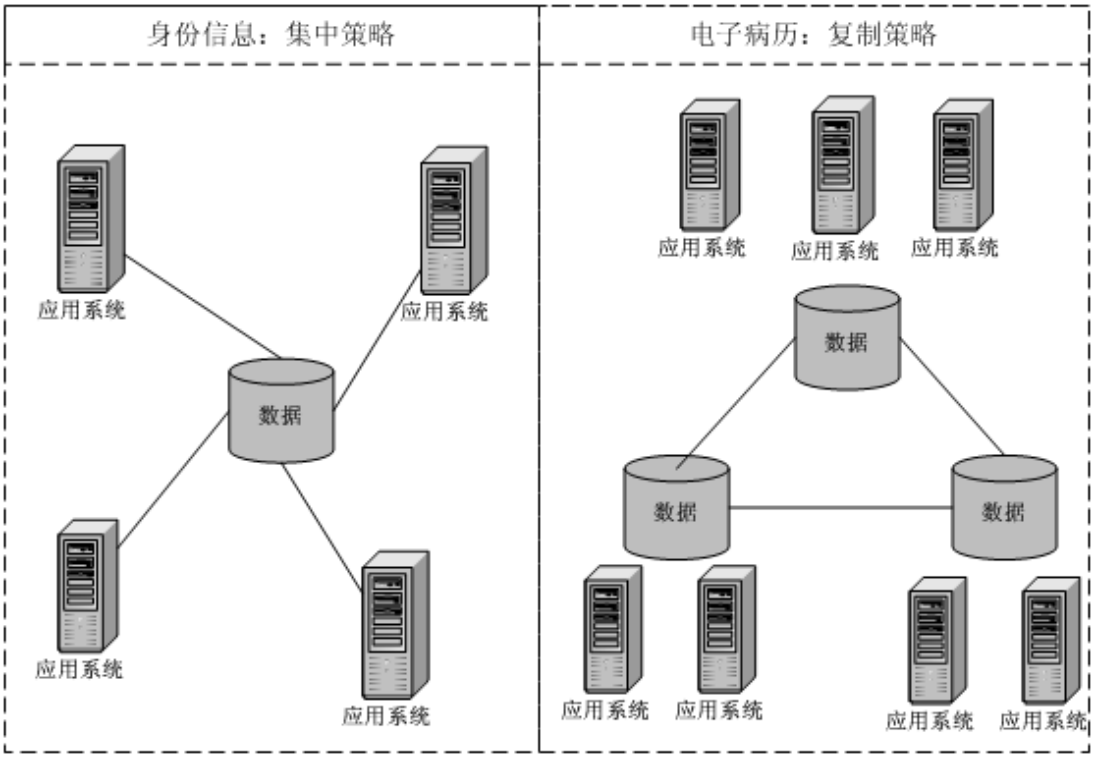


图 15-10　把握系统特点，确定分布策略（合适原则）

3.3 综合原则

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，就需要考虑综合运用不同数据分布策略。

3.4 案例：服务受理系统 vs. 外线施工管理系统

案例背景

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 15-11 所示，为客户申请服务开通所涉及的业务流程。

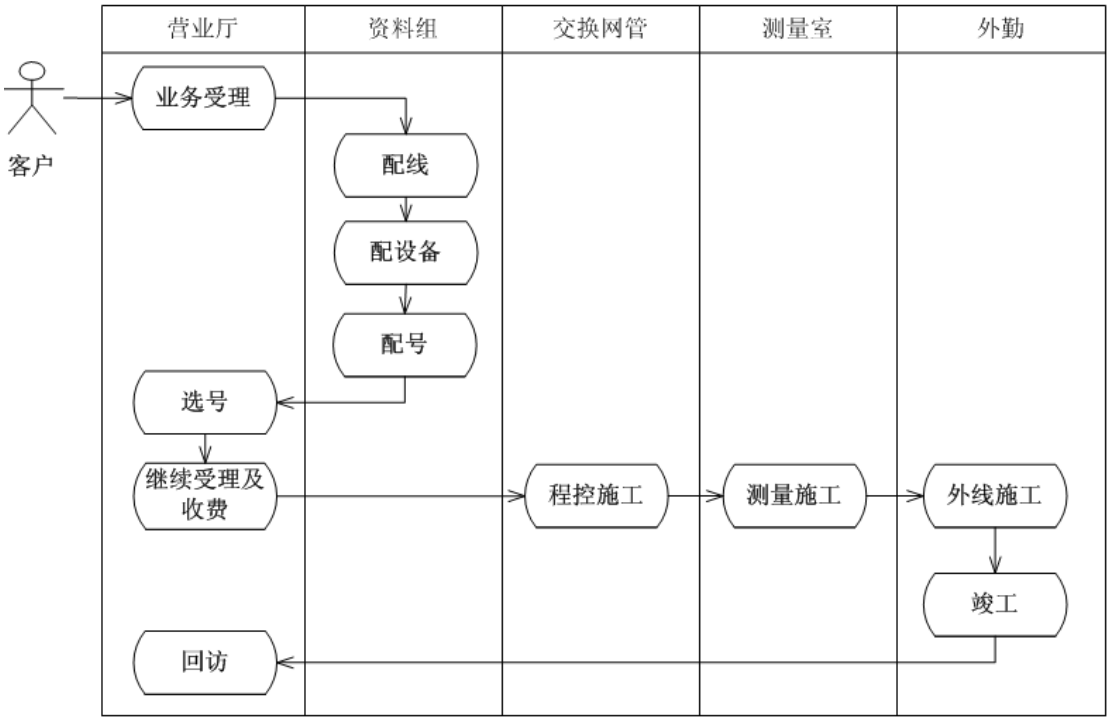


图 15-11 客户申请服务开通所涉及的业务流程

数据分布方案

其中的“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，各有个的业务数据，于是采用“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“□□市电信服务受理系统”而言，应采用什么数据分布策略呢？考虑系统欲达到的以下目标：

- 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 15-12 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

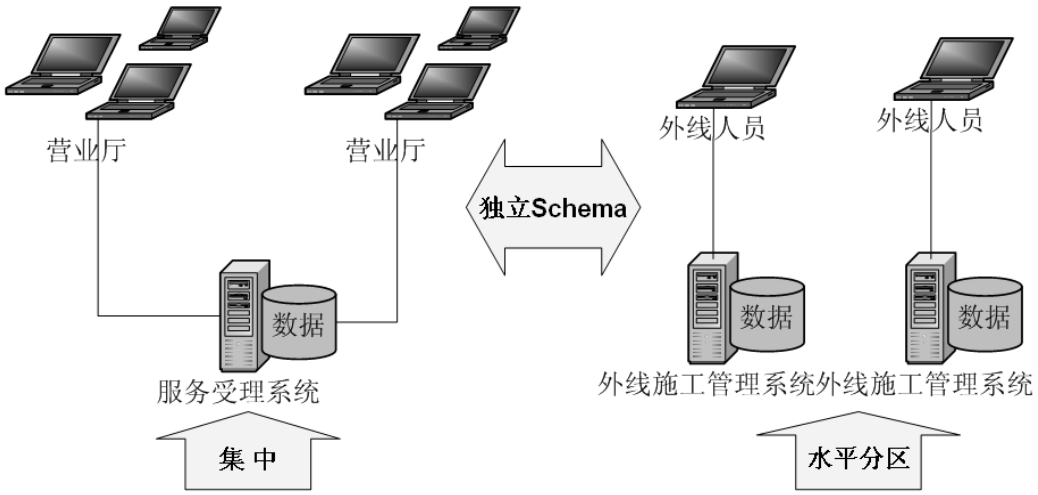


图 15-12　不同分布策略，可以综合运用（综合原则）

3.5 优化原则

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。当难以一步到位地做出数据分布策略的正确选择、当还存在质疑的时候，应从“对吗”、“好吗”两方面进行对比、评估、优化。

3.6 案例：铃声下载门户