软件架构设计 第2版

程序员向架构师转型必备

客户企业反馈版

温 昱 著
昱培咨询 策划

目 录

第 1 章　从程序员到架构师

> 自由竞争越来越健全，真正拥有实力的人越来越受到推崇。……努力钻研，力求在更高水平上解决问题的专家不断增加，这正如电脑处理信息的能力在不断提高一般。如今，这样的时代正在到来。
>
> ——大前研一，《专业主义》

> 机会牵引人才，人才牵引技术，技术牵引产品，产品牵引更大的机会。在这四种牵动力中，人才所掌握的知识处于最核心的地位。
>
> ——张利华，《华为研发》

人才、和合理的人才结构，是软件公司乃至软件业发展的关键。

成才、并在企业中承担重要职责，是个人职业发展的关键。

1.1　软件业人才结构

1.1.1　金字塔型，还是橄榄型？

有人说，软件业当前的人才结构是橄榄型（中间大两头小），需求量最大的“软件蓝领”短缺问题最为凸显，这极大地制约着软件业的发展，因此要花大力气培养大量的初级软件程序员等“蓝领工人”。

但业内更多人认为，软件业当前的人才结构是金字塔型，高手和专家型人才的总量不足才是“制约发展”的要害，因此一方面软件工程师应争取提升技能、升级转型，另一方面企业和产业应加强高级技能培训、高级人才培养。

软件业的人才结构，到底是金字塔型，还是橄榄型？

本书认为，一旦区分开“学历结构”和“能力结构”，问题就不言自明了（如图 1-1 所示）：

- 学历结构 ＝ 橄榄型。“中级学历”最多。有资料称，软件从业者中研究生：本科：专科的比例大致是 1:7:2。
- 能力结构 ＝ 金字塔型。“初级人才”最多。工作 3 年以上的软件工程师，就一跃成为“有经验的中级人才”了吗？显然不一定。
- 有学历 ≠ 有能力。每个开发者真正追求的，是成为软件业“人才能力结构”的顶级人才或中级人才。

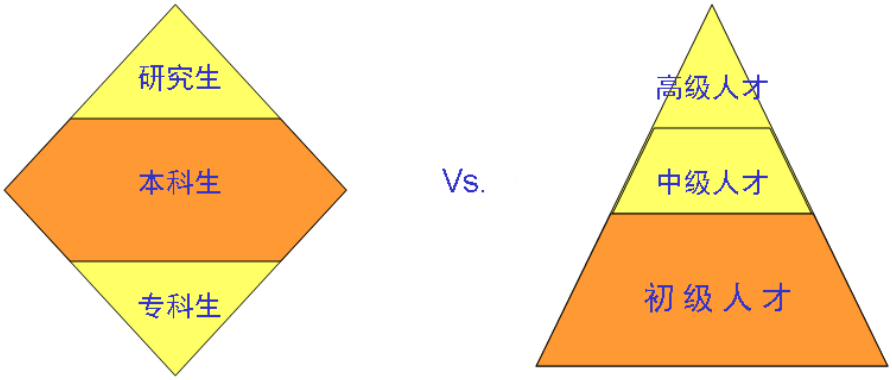


图 1-1　人才结构的两个视角

1.1.2　从程序员向架构师转型

人才能力的金字塔结构，注定了软件产业的竞争从根本上是人才的竞争。具体到软件企业而言，一个软企发展的好坏，极大地取决于如下人才因素：

- 员工素质
- 人才结构
- 员工职业技能的纵深积累
- 员工职业技能的适时更新

借用《华为研发》一书中的说法，“机会、人才、技术和产品是公司成长的主要牵动力。机会牵引人才，人才牵引技术，技术牵引产品，产品牵引更大的机会。在这四种牵动力中，人才所掌握的知识处于最核心的地位。”

然而，纯粹靠从外部“招人”，不现实。何况，软件企业、软企的竞争对手和软件产业环境，都处在动态发展之中。因此，软件企业应该：

- 定期分析和掌握本公司的员工能力状况、人才结构状况
- 员工专项技能的渐进提升（例如架构技能、设计重构技能）
- 研发骨干整体技能的跨越转型（例如高级工程师向架构师、系统工程师和技术经理的转型）

对于本书的主题“软件架构设计能力的提升”而言，架构设计能力是实践性很强的一系列技能，从事过几年开发工作是掌握架构设计各项技能的必要基础。因此可以说，“从程序员向架构师转型”不仅是软件开发者个人发展的道路之一，也是企业获得设计人才的合适途径。

1.2 本书价值

本书包含 3 部分，分别是：

- 第 1 部分：基础概念篇
- 第 2 部分：实践过程篇
- 第 3 部分：模块划分专题

读者可以根据自身发展状况、实际工作需要，选择合适的阅读路径（如图 1-2 所示）。

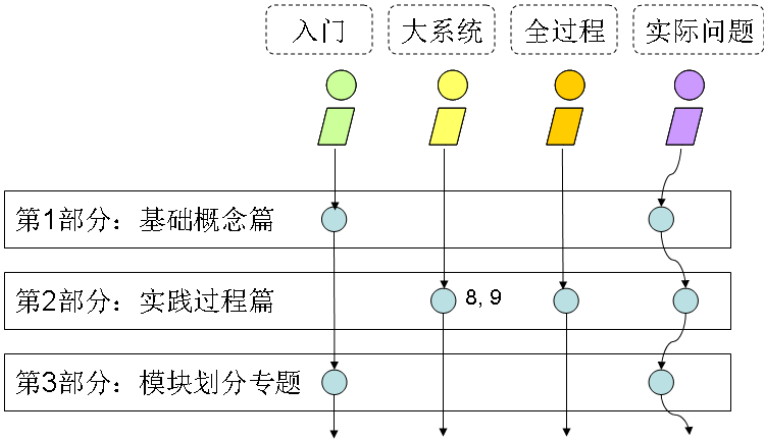


图 1-2　不同目的，不同阅读路径

1.2.1 阅读路径 1：架构设计入门

对于架构还未入门的程序员，推荐先重点阅读“基础概念篇”和“模块划分专题”：

- 基础概念篇
  解析架构概念（第 2 章）之后，讲解如何运用“逻辑视图+物理视图”设计架构（第 3 章）。
- 模块划分专题
  讲解模块划分的不同方法，功能模块、分层架构、用例驱动的模块划分过程等内容将被讨论（第 12-15 章）。

架构设计入门必过“架构视图关”。本书细致讲解“逻辑视图＋物理视图”的运用，如图 1-3 所示。体会了“分而治之”和“迭代式设计”这两点关键思想，运用“逻辑视图＋物理视图”设计一个系统的架构也就不那么难了。

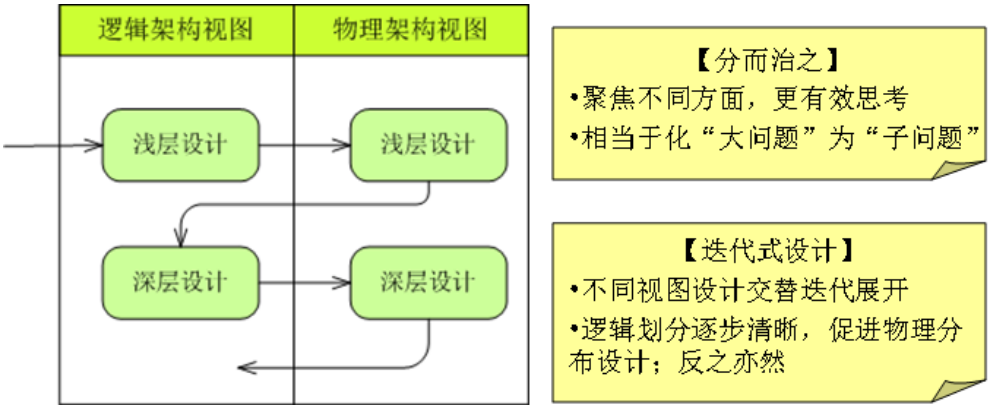


图 1-3　两视图法的“分而治之”和“迭代”思想

架构设计入门必过“模块划分关”。本书“模块划分专题”总结了模块划分的 4 种方式，如图 1-4 所示。为企业所做的多次培训与交流一再表明，“水平分层”、“垂直划分功能模块”和“从用例到类、再到模块”的模块设计思想正是软件设计者实际工作当中不断应用的方法（不推荐的想到哪“切”到哪的设计方式也大量存在）。

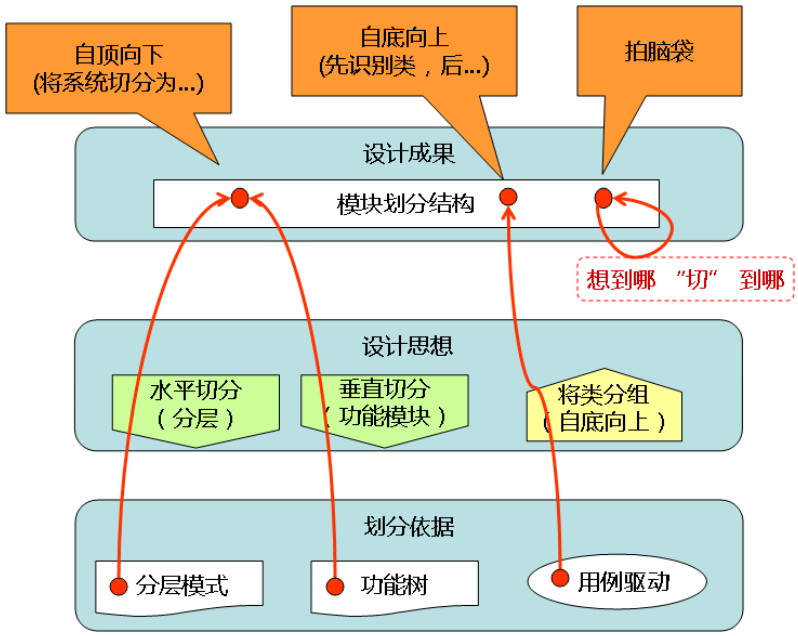


图 1-4　业界模块划分的 4 种做法

……当一个程序员，看懂了他们团队的架构师是如何划分模块的（甚至问“你为啥只垂直切子系统没分层呢”），那他离成为架构师还远吗？

1.2.2　阅读路径 2：领会大系统架构设计

入门之后，进一步学习大型系统架构成败的关键——概念架构设计。推荐精读如下两章：

- 第 8 章。如何确定影响架构设计的关键需求。
- 第 9 章。概念架构如何设计。

小系统和大系统的架构设计之不同，首先是“概念架构”上的不同，而归根溯源这是由于架构要支撑的“关键需求”不同造成的。整个架构设计过程中的“确定关键需求”这一环节，可谓小系统和大系统架构设计的“分水岭”，架构设计走向从此大不同。

概念架构是直指系统目标的设计思想、重大选择，因而非常重要。《方案建议书》、《技术白皮书》和市场彩页中，都有它的身影，以说明产品/项目/方案的技术优势。也因此，有人称它为“市场架构”。

概念架构设计什么？从设计任务上，概念架构要明确“1 个决定、4 个选择”，如图 1-5 所示。

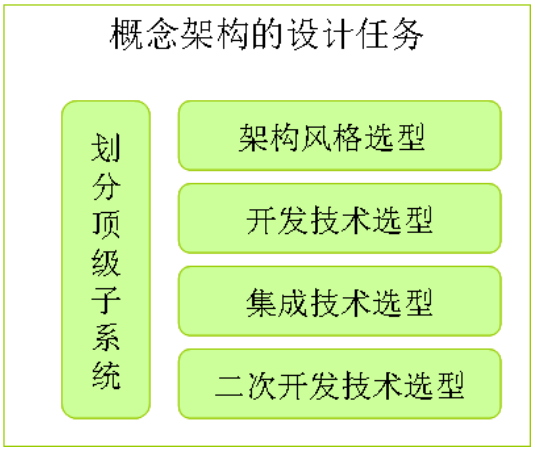


图 1-5　概念架构设计什么？

概念架构如何设计？从设计步骤的顺序上，本书推荐（如图 1-6 所示）：

- 首先，选择架构风格、划分顶级子系统。这 2 项设计任务是相互影响、相辅相成的。
- 然后，开发技术选型、集成技术选型、二次开发技术选型。这 3 项设计任务紧密相关、同时进行。另外可能不需要集成支持，也可以决定不支持二次开发。

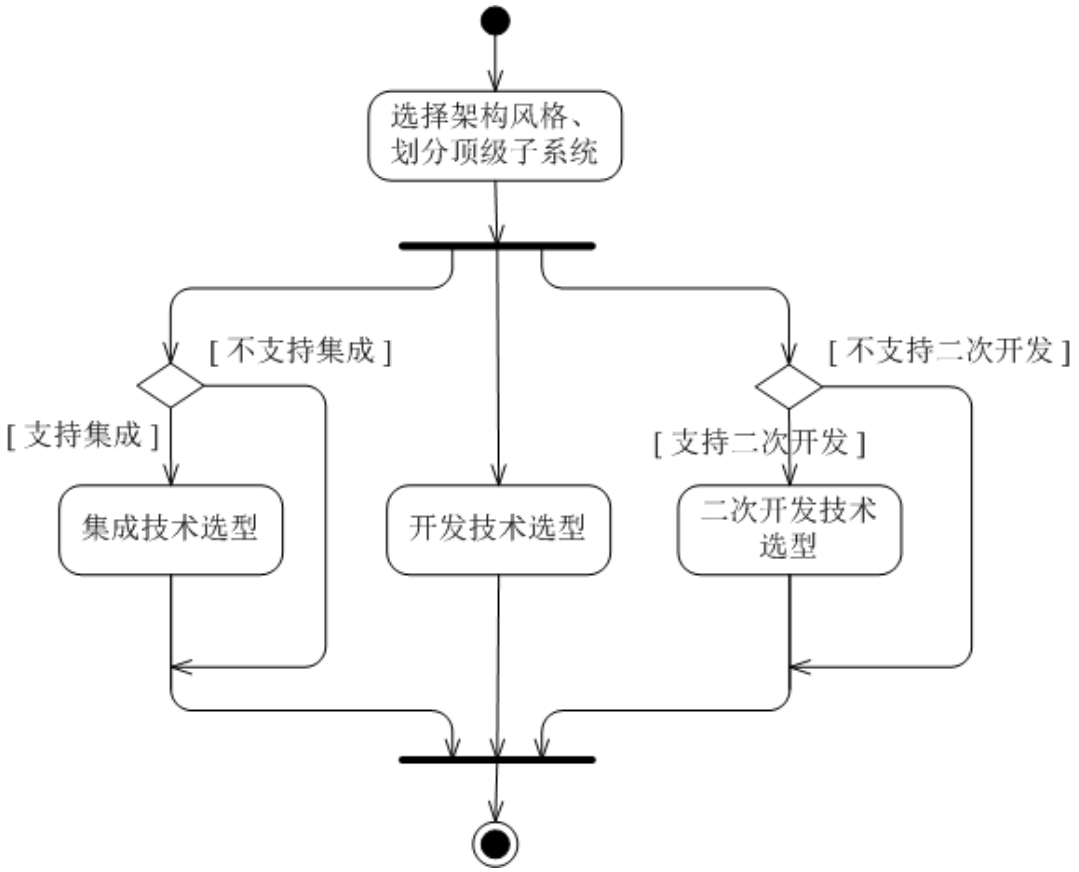


图 1-6　概念架构如何设计？

1.2.3　阅读路径 3：从需求到架构的全过程

专业的架构设计师，必须掌握架构设计的“工程化过程”。以此为目标的读者，请精读“实践过程篇”的内容：

- 第 4 章。概述从需求到架构的全过程，还提供了一节叫“速查手册”，供快速查阅每个环节的工作内容。
- 第 5-11 章。如下 6 个环节的展开讨论：需求分析、领域建模、确定关键需求、概念架构设计、细化架构设计、架构验证。

内容虽多，用一幅图概括也不是不可能，在为企业培训架构时我们就经常这么做——如图 1-7 所示，展示了从需求到架构整个过程中的关键任务项。

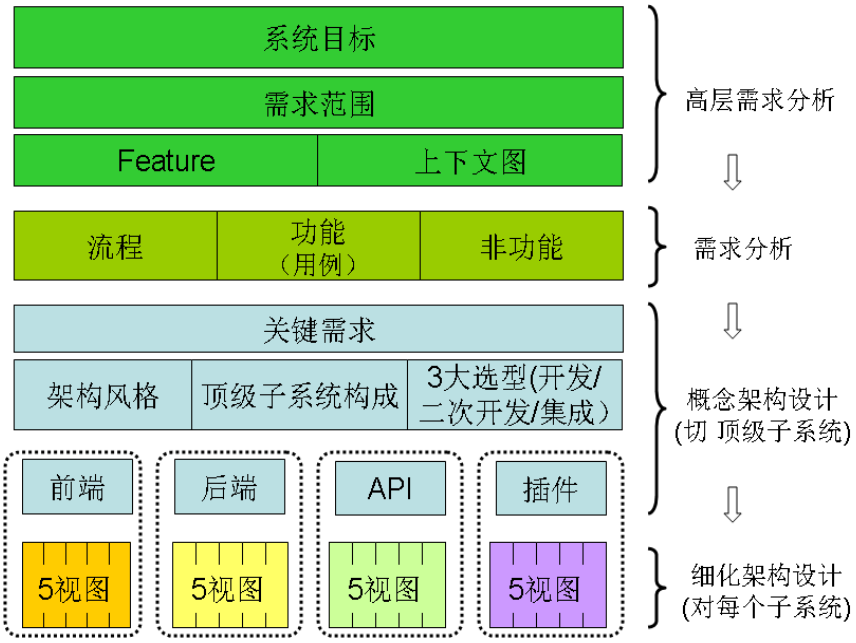


图 1-7　架构设计的全过程

1.2.4　阅读路径 4：结合工作，解决实际问题

最后，按照经典名著《如何阅读一本书》中所说的“主动阅读”法，读者还可以“带着问题”、“以我为主”地“跳读”。

【实际问题 1】很典型地，开发人员的一个经典困惑是：领域经验不足怎么办？

本书第 7 章，给出了建议。破解“领域知识不足”死结的一个有效方法，是把领域模型作为“理解领域的手段”。领域模型的“强项”是“理顺概念关系、搞清业务规则”——通过对复杂的领域进行“概念抽象”和“关系抽象”建立模型、获得对领域知识总体上的把握，就不会掉入杂乱无章的概念“堆”里了。

【实际问题 2】又例如，你所在的部门，是否存在这样的问题：不同的人对架构有不同的理解？

本书第 2 章，推荐结合部门的实际工作，来理解架构的含义，统一团队不同成员对架构的理解。

【实际问题 3】用例建模够不够？流程建模要不要？笔者接触的很多实践者，都有此困惑。

本书第 6 章，简述了需求分析“三套实践论”的观点，提出“大、中、小”三套可根据系统特点选择的需求实践策略，可供实践者参考。如图 1-8、图 1-9 和图 1-10 所示。

图 1-8　需求实践论之小型方法

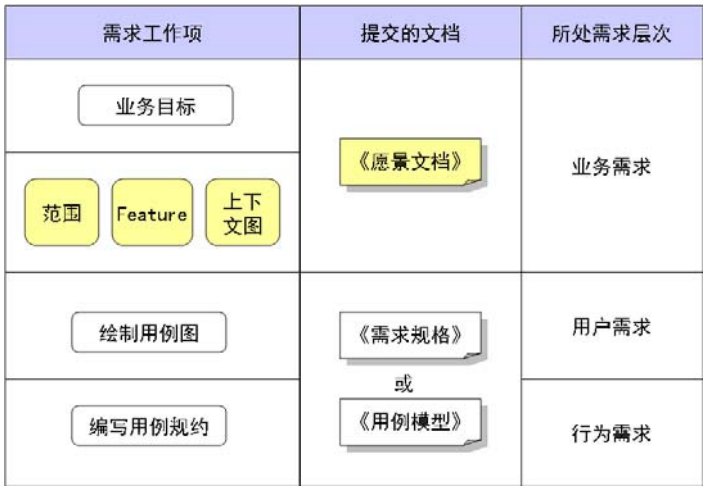


图 1-9　需求分析实践论之中型方法

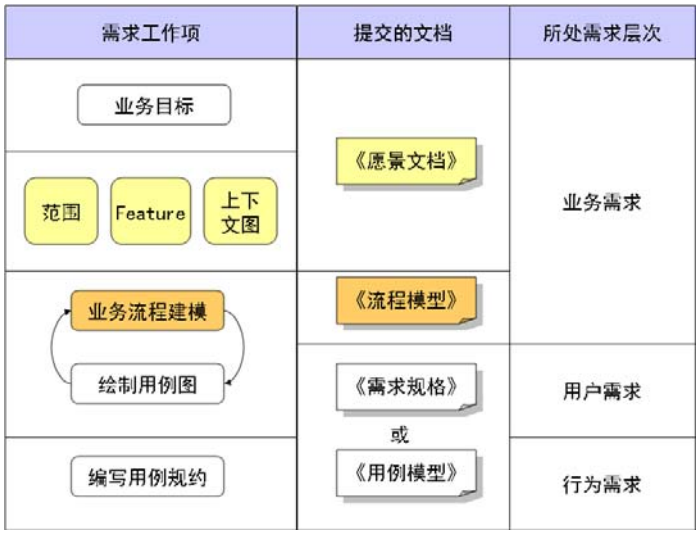


图 1-10　需求实践论之大型方法

由此也可见，是实践决定方法，不是方法一刀切实践。

更多问题的解决思路在此不再列举，祝阅读之旅愉快！

第 3 章　理解架构设计视图

> “有角度就有空间”。……多视图方法背后的核心思想有些相似：从不同角度，规划“分割”与“交互”。
>
> ——温昱，《一线架构师实践指南》

> 不同的视图支持不同的目标和用途。
>
> ——Paul Clements，《软件构架编档》

架构设计是一门解决复杂问题的艺术。

设计任何复杂系统时，架构视图都是不可或缺的（无一例外）。但由于在日常开发工作中较少接触，大部分程序员对“设计视图”的思想还比较陌生。

本章围绕“架构视图”这一主题，将逐次讨论：

- 设计架构时，架构视图为什么必不可少？【本章问题 1】
- 什么是架构视图？【本章问题 2】
- 如何运用“逻辑视图＋物理视图”设计一个系统的架构？【本章问题 3】

3.1　软件架构为谁而设计

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需要编写哪些 C 程序或 OO 类、使用哪些现成的程序库（Library）和框架（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以及编译过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，其他一切只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，软件架构应该将系统划分为一个个功能子系统。程序员又笑了。

大家想了想说，我们都没错呀，所有这些方面都是需要的啊。……软件架构师哭了。

由此看来，不同涉众看待架构的视角是不同的，而架构师要为不同的涉众而设计。

怎么办呢？同时关注多个架构设计视图。架构视图的本质是“分而治之”，能帮助架构师从不同角度设计，特别是面对复杂系统时“分而治之”地设计是必须的。

这就回答了“设计架构时，架构视图为什么必不可少”的问题【本章问题 1】。

3.1.1 为用户而设计

为什么要开发某个软件系统呢？因为要给用户使用：或辅助用户完成日常工作，或帮助用户管理某些信息，或给用户带来娱乐体验……不一而足。

用户要功能，用户也要质量。

每套软件都会提供这样或那样的功能，正是这些功能帮助用户实现他们在工作或生活中的特定目标。用户所需的功能和系统本身的结构一定是相互影响的，这正是软件架构师要特别关注的。先举个生活中的例子吧。例如，因为功能不尽相同，所以“转笔刀”的结构和“专用刀片”的结构也不一样（如图 3-1 所示）：小学生需要削铅笔，那一定是转笔刀最为适合他们，因为方便易用，一转即可；而美术师需要用铅笔来画素描，他们则需要使用专用刀片，因为转笔刀难以削出满足不同绘画要求的形状各异的笔尖。同样，对于软件系统而言，用户需求是千差万别的，我们采用的软件架构必须和所要提供的功能相适应。因此，软件架构师必须时时牢记：为用户而设计。

图 3-1　生活中的例子：功能与结构相互影响

诸如性能、易用性等软件质量属性，并不像上面所述的软件功能那样直接帮助用户达到特定目标，但并不意味着软件质量属性不是必需的——恰恰相反，质量属性差的软件系统大多都不会

成功。例如，你提供了用户要求的“交易查询”功能，但这一功能动辄就需要花上几分钟，用户能接受吗？当然不能接受。用户会说，功能虽然具备了，但质量太差难以接受。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、持续可用性和鲁棒性等。因此，软件架构师也应当时时牢记：为用户而设计，不仅满足用户要求的功能，也要达到用户期望的质量。

3.1.2 为客户而设计

很多时候，客户（Customer） ≠ 最终用户（User）。

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。

架构师为客户而设计：充分考虑客户的业务目标、上线时间的要求、预算限制，以及集成需要等，还要特别关注客户所在领域的业务规则和业务限制。为此，架构师应当直接或（通过系统分析员）间接地了解和掌握上述需求及约束，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。合适的才是最好的，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。

3.1.3 为开发人员而设计

先研究需求，再设计架构，然后交由开发人员编程……。作为架构，不仅要为“上游”的为需求而设计，还要为“下游”的开发人员而设计。

例如，性能是软件运行期质量属性，最关心性能的人其实是客户；但可扩展性是软件开发期质量属性，项目开发人员和负责升级维护的开发人员最关心。

推及开去，其实，并不是所有需求都来自用户，软件的可扩展性、可重用性、可移植性、易理解性和易测试性等非功能需求更多地考虑对开发人员的影响。这类“软件开发期质量属性”深刻影响开发人员的工作，使开发更顺畅抑或更艰难。

3.1.4 为管理人员而设计

软件变得越来越复杂，单兵作战不再普遍，取而代之的是团队开发。而团队开发又反过来使软件开发更加复杂，因为现在不仅仅要面临技术复杂性的问题了，还有管理复杂性的问题。

开发人员之间的依赖，源自他们负责的程序之间的依赖（如图 3-2 所示）。要理清并管理人员协作，就应该搞清楚系统一级“模块+交互”的设计、搞清楚架构。可见，架构是开发管理的核心基础。

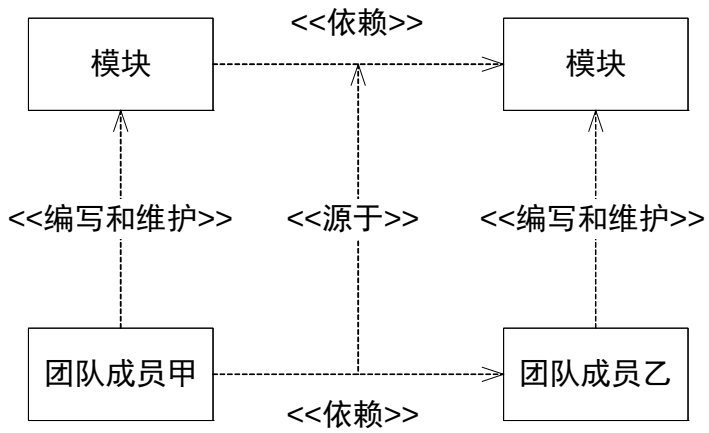


图 3-2　开发人员之间的依赖，源自他们负责的程序之间的依赖

看来，软件架构师也应为管理人员而设计。

例如，对软件项目管理而言，软件架构应当起到应有的作用：为项目经理制定项目计划、管理项目分工和考核项目进度等提供依据。一方面，软件架构从大局着手，就技术方面的重大问题作出决策，构造一个由粗粒度模块组成的解决方案，从而可以把不同模块分配给不同小组分头开发。另一方面，软件架构设计方案规定了各模块之间如何交互的机制和接口，在开发小组之间起到“沟通桥梁”和“合作契约”的作用。

再例如，对软件配置管理而言，软件架构师亦应顾及。一般而言，在软件架构确定之前，软件配置管理是无法全面开展的。配置管理员应该能够从软件架构方案中了解到开发出来的软件以什么样的目录结构存在，以及编译过后的软件目标模块放到哪个目录等决定，并以此作为制定配置管理基本方案的基础。

3.1.5　总结

架构师应当为项目相关的不同角色而设计（如图 3-3 所示）——只有这样，软件架构才能和它“包含了关于如何构建软件的一些最重要的设计决策”的“地位”相符。

- 架构师要为“上游”客户负责，满足他们的业务目标和约束条件；
- 架构师要为“上游”用户负责，使他们关心的功能需求和运行期质量属性得以满足；
- 架构师必须顾及处于协作分工“下游”的开发人员；
- 架构师还必须考虑“周边”的管理人员，为他们进行分工管理、协调控制和评估监控等工作提供清晰的基础。

图 3-3　软件架构为谁而设计

3.2　理解架构设计视图

什么是架构视图呢？【本章问题 2】

3.2.1　架构视图

架构视图的实践导向性很强，每个视图分别关注不同的方面，针对不同的实践目标和用途。Philippe Kruchten 在其著作《Rational 统一过程引论》中写道：

> 一个架构视图是对于从某一视角或某一点上看到的系统所作的简化描述，描述中涵盖了系统的某一特定方面，而省略了与此方面无关的实体。

架构视图是一种设计架构、描述架构的核心手段：

- 也就是说，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，也为软件架构的理解、交流和归档提供了方便。

在多种架构视图中，最常用的是逻辑架构视图和物理架构视图（本章稍后讨论如何运用“逻辑视图+物理视图”设计架构）。

3.2.2 一个直观的例子

在讲解软件的逻辑架构视图和物理架构视图之前，先看一个生活中视图的例子。

图 3-4 所示的世界人口分布图，是社会学家关心的；而气候学家，则更关心图 3-5 所示的世界年降水量分布图。这其实就运用了“视图”的手段，用不同视图来刻画我们关心的世界的不同方面——你完全可以将“世界人口分布图”称为“世界的人口分布视图”。

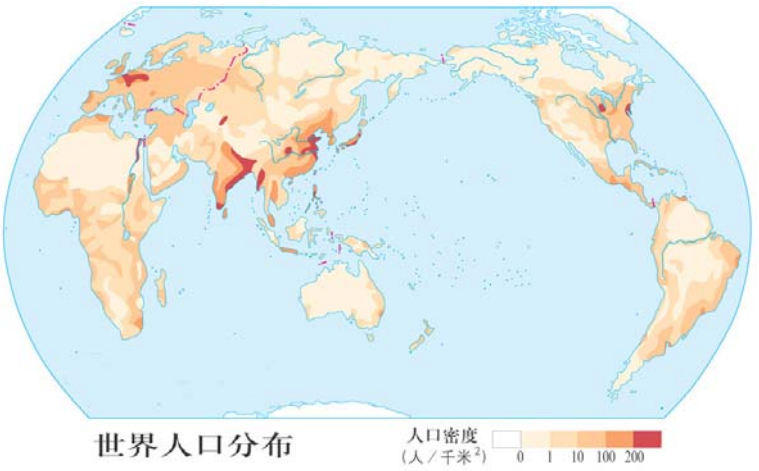


图 3-4 世界人口分布图（图片来源：www.dlpd.com）

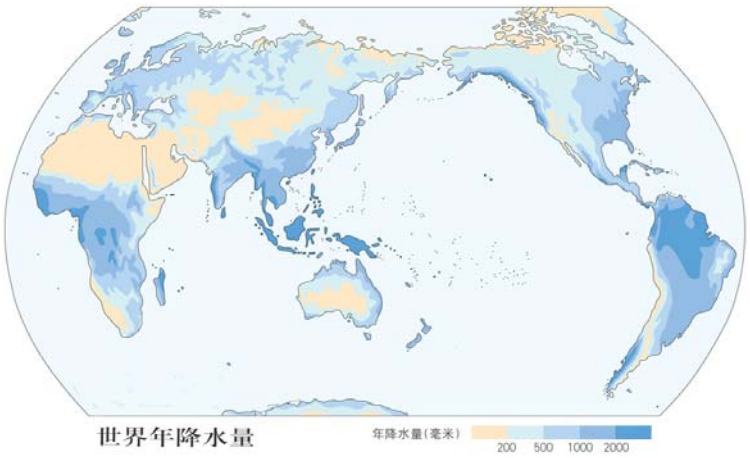


图 3-5　世界年降水量分布图（图片来源：www.dlpd.com）

当然，更进一步而言，同一事物的不同视图之间是有联系的。例如，从图 3-4 和 3-5 上看，除了南美洲之外基本都是降水量足的地方人口较密集。我们下面将讨论的软件架构视图也是这样，不同视图之间存在相互影响。

3.2.3　多组涉众，多个视图

首先，考虑软件架构的表达与交流。

这是个很现实的问题。究其原因，由于角色和分工不同，整个软件团队以及客户等涉众各自需要掌握的技术或技能存在很大差异，为了完成各自的工作，他们需要了解整套软件架构决策的不同子集。所以，软件架构师应当提供不同的软件架构视图，以便交流和传递设计思想。例如：

- 系统工程师：由于负责部署和运营维护，他们最关心软件系统基于何种操作系统之上、依赖于哪些软件中间件、有没有群集或备份等部署要求、驻留在不同机器上的软件部分之间的通信协议是什么等决策；
- 而开发人员：则最关心软件架构方案中关于模块划分的决定、模块之间的接口如何定义、甚至架构指定的开发技术和现成框架是不是最流行的等问题；
- ……不一而足。

相反，如果不同视图混为一谈，《架构文档》理解起来会非常困难、甚至看不懂。对此，Peter Herzum 等人在《Business Component Factory》一书中曾指出：

> 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)。

第二，考虑软件架构的设计。

这个问题更为关键。软件架构是个复杂的整体，架构师可以“一下子”把它想清楚吗？当然不能。有关思维的科学研究表明，越是复杂的问题越需要分而治之的思维方式：

- 通过“架构视图”作为分而治之的手段，使架构师可以分别专注于架构的不同方面、

相对独立地分析和设计不同“子问题”，这相当于将“整个问题”简化和清晰化了；

- 再也不必担心逻辑层（Layer）、物理层（Tier）、功能子系统、模块、接口、进程、线程、消息和协议等设计统统混为一“潭”（泥潭？）了。

不幸的是，大多数书籍中都强调多视图方法是软件架构归档的方法，却忽视了该方法对架构设计思维的指导作用。而本书强调，多视图不仅是归档方式，更是架构设计的思维方式。

3.3 运用“逻辑视图+物理视图”设计架构

无论是软件架构、还是现实生活，运用“逻辑视图 + 物理视图”刻画大局，都方便有效。图 3-6 所示为办公室局域网：从物理角度看，所有计算机“毫无区别”地连接到路由器上；而从逻辑角度看呢，一台计算机充当文件服务器，而其他计算机是可以访问服务器的客户机。

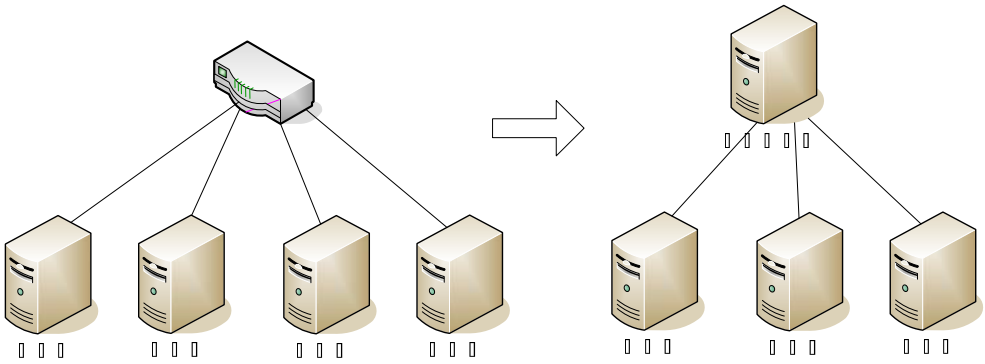


图 3-6　区分物理视角与逻辑视角

那么，如何运用“逻辑视图 + 物理视图”设计一个系统的架构呢？【本章问题 3】

3.3.1 逻辑架构

软件的逻辑架构规定了软件系统由哪些逻辑元素组成以及这些逻辑元素之间的关系。具体而言，组成软件系统的逻辑元素可以是逻辑层（Layer）、功能子系统、模块。

设计逻辑架构的核心任务，是比较全面地识别模块、规划接口，并基于此进一步明确模块之间的使用关系和使用机制。图 3-7 展示了一个网络设备管理系统逻辑架构设计的一部分，“模块+接口”是其设计的基本内容：

- 识别模块（例如，图中的 TopoGraphModel 等）
- 规划模块的接口（例如，图中的 StatusChangedRender 等）

- 明确模块之间的使用关系和使用机制（例如，图中的 StatusChanged 是由 GeneralModel 创建的消息，以参数的形式传递给 StatusChangedRender 接口）

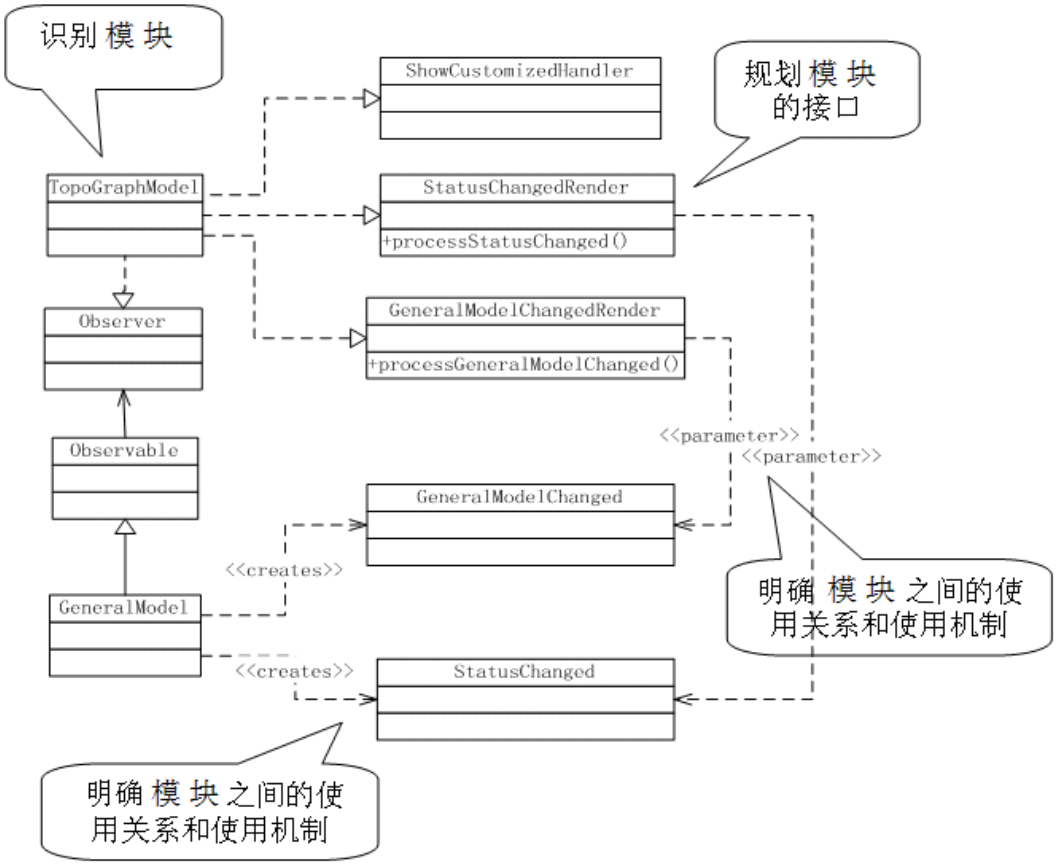


图 3-7　软件逻辑架构设计的核心任务

3.3.2　物理架构

软件的物理架构规定了组成软件系统的物理元素，这些物理元素之间的关系，以及它们部署到硬件上的策略。

物理架构可以反映出软件系统动态运行时的组织情况。此时，上述物理架构定义中所提及的“物理元素”就是进程、线程以及作为类的运行时实例的对象等，而进程调度、线程同步、进程或线程通信等则进一步反映物理架构的动态行为。

随着分布式系统的流行，“物理层（Tier）”的概念大家早已耳熟能详。物理层和分布有关，通过将一个整体的软件系统划分为不同的物理层，可以把它部署到分布在不同位置的多台计算机上，从而为远程访问和负载均衡等问题提供了手段。当然，物理层是大粒度的物理单元，它最终

是由粒度更小的组件、模块和进程等单元组成的。

物理架构的应用很广泛。例如，架构设计中可能需要专门说明数据是如何产生、存储、共享和复制的，这时可以利于物理架构，展示软件系统在运行期间数据是由哪些运行时单元产生以及如何产生的，数据又如何被使用，如何被存储，哪些数据需要跨网络复制和共享等方面的设计决策。

由此可见，对于“逻辑架构 + 物理架构”这个“2 视图方法”而言，组成软件系统的“物理元素”涉及的内容比较宽泛，如图 3-8 所示。（也正是因为这个原因，设计大型系统的架构时会引入更多架构设计视图，从而使每个视图的设计内容更加明确，本书后续要讲的 5 视图法包含了逻辑架构、开发架构、运行架构、数据架构和物理架构。）

图 3-8 “2 视图方法”的物理架构可能的设计内容

3.3.3 从“逻辑架构+物理架构”到设计实现

架构设计，指导后续的详细设计和编程；而详细设计和编程，贯彻和利用这些设计（如图 3-9 所示）：

- 逻辑架构中关于职责划分的决策，体现为层、功能子系统和模块等的划分决定，从静态视角为详细设计和编程实现提供切实的指导；有了分解就必然产生协作，逻辑架构还规定了不同逻辑单元之间的交互接口和交互机制，而编程工作必须实现这些接口和机制。
- 所谓交互机制，是指不同软件单元之间交互的手段。交互机制的例子有：方法调用、

基于 RMI 的远程方法调用、发送消息等。

- 至于物理架构，它关注的是软件系统在计算机中运行期间的情况。物理架构设计视图中规定了软件系统如何使用进程和线程完成期望的并发处理，进程线程这些主动单元（Active Unit）会调用哪些被动单元（Passive Unit）参与处理，交互机制（如消息）为何等问题，从而为详细设计和编程实现提供切实指导。

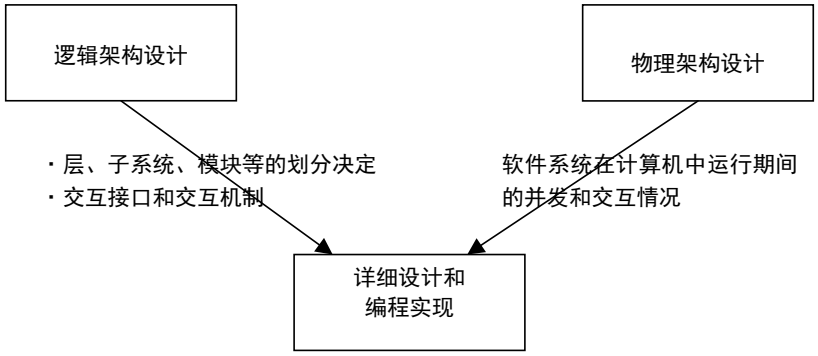


图 3-9　逻辑架构和物理架构对后续开发的作用

3.4　实际应用（2）——开发人员如何快速成长

3.4.1　开发人员应该多尝试设计

你们公司，你所在的部门，发现了一个问题，即“兵多将少”。程序员足够多但设计人员太少。

设计不满意似乎成了常态……

诸如相似模块的低水平重复开发等困扰挥之不去……

部门的总体工作效率低下……

你们会也开了，问题也讨论了，问题背后的问题也被“揪”出来了——开发人员的成长速度低于预期。

怎么办呢？

如果你是程序员，又盼着有一天能做架构师，就要明白“经历设计在前，成为架构师在后”的道理。也就是说，程序员要尽量找机会多看看别人的设计成果、多体会别人的设计过程、多试着自己来设计设计——这些方式都能促进对设计方法或技巧的领会。

如果你是主管，又深感手下“兵多将少”设计人才短缺，就要确认一下是不是给程序员接触设计的机会太少了。也就是说，主管应该尽量创造机会让程序员多看看别人的设计成果、多体会别人的设计过程、多试着自己来设计设计。

笔者的经验表明，结合培训、让开发人员对实验项目进行设计，也不失为一种快速提升设计技能的好办法。

3.4.2 实验项目：案例背景、训练目标

下面，我们来设计一个名为 MailProxy 的邮件代发系统。

众多公司的“客户服务系统”都需要批量地向客户发送邮件。(“客户服务系统”管理着企业对客户的服务内容，包括客户投诉、故障处理、客户咨询、客户查询、客户回访、客户建议、客户关怀等服务信息以及服务指标信息等。）而 MailProxy 作为一款软件产品，其核心功能就是：邮件代发。图 3-10 所示的用例图刻画了 MailProxy 的基本功能：

- MailProxy 和客户系统对接
- 通过对 Mail Server 系统的调度完成自动邮件代发
- 反馈发送结果给客户系统（含 Log 日志等）
- 还提供灵活的规则设置等功能

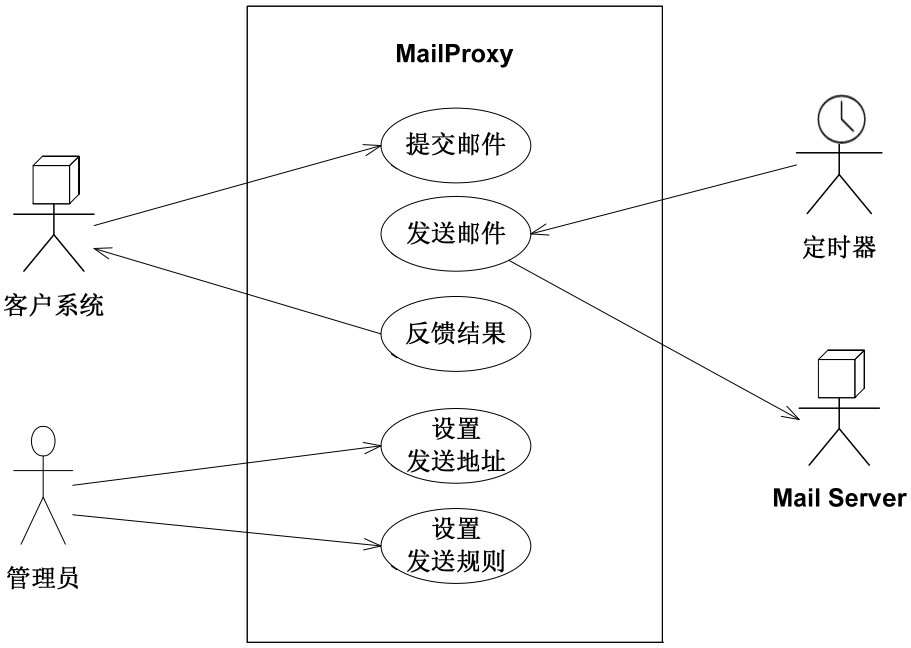


图 3-10　MailProxy 系统的用例图

“训练”目标：

- 运用本章所讲的“逻辑视图+物理视图”技能
- 辅以分而治之、迭代式设计两个技巧
  - 【分而治之】。逻辑架构进行“逻辑分解”，物理架构进行“物理分解”，它们分别关注架构的不同方面，利于思维聚焦。（相当于化“大问题”为“子问题”）
  - 【迭代式设计】。不同架构视图的设计交替进行、迭代展开。逻辑职责的划分逐步清晰，促进了物理分布设计；反之亦然。（死抠“未知”抠不出来，就利用当前“已知”探索更多“未知”，循环着来嘛）

3.4.3　逻辑架构设计（迭代 1）

……深入研究需求之后，发现 MailProxy 系统的一个显著特点：不仅要和“人”交互，更要

和多种“外部系统”交互。

因此，设计逻辑架构时，除了包含用户交互层、业务逻辑层、数据访问层之外，一定应包含一个“系统交互层”（如图 3-11 所示）。

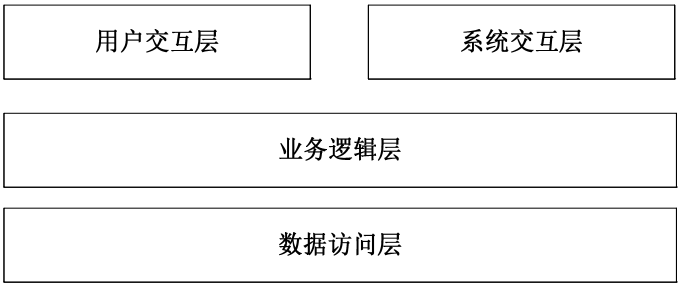


图 3-11　着手设计逻辑架构：分层

3.4.4　物理架构设计（迭代 1）

多视图≠多阶段，不应瀑布式地设计每个视图，而是应该迭代式设计。而且，对大型系统而言，先把逻辑架构详尽地设计完再开始物理架构的设计，有很大的问题：

- 一是比较困难。毕竟，逻辑架构、物理架构是同一系统的不同方面；如果系统比较复杂，在对一个方面的设计毫无所知的情况下，过于深入设计的另一方面太过盲目了。
- 二是不利于提高设计质量。而逻辑架构、物理架构交替进行，本身就是不断相互验证、促进设计优化的过程。

应该早些开始物理架构视图的设计。现在，着手设计 MailProxy 的物理分布，如图 3-12 所示。

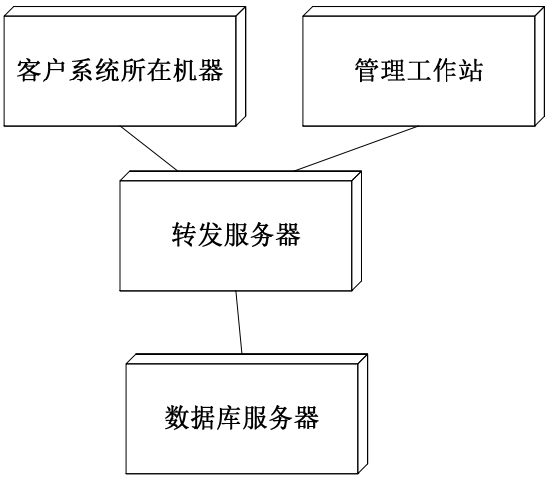


图 3-12　着手设计物理架构：基本的物理分布

3.4.5　逻辑架构设计（迭代 2）

原先接手 MailProxy 时，对它相当陌生。

设计逻辑架构时，确立了基本分层结构之后就感觉后续设计“展不开”、“细不下去”。

现在，有了初步的逻辑分层、还有基本的物理分布设计，转回头来再继续逻辑架构设计试试。……嗯，根据物理架构的提示，逻辑层“系统交互层”中应该有“Mail Server 交互”模块吧，它封装和具体 Mail Server 的交互。……嗯，“用户交互层”应当有“设置地址”、“设置规则”等 UI 模块。

逐步细化逻辑架构的设计，得到图 3-13 所示的逻辑架构。

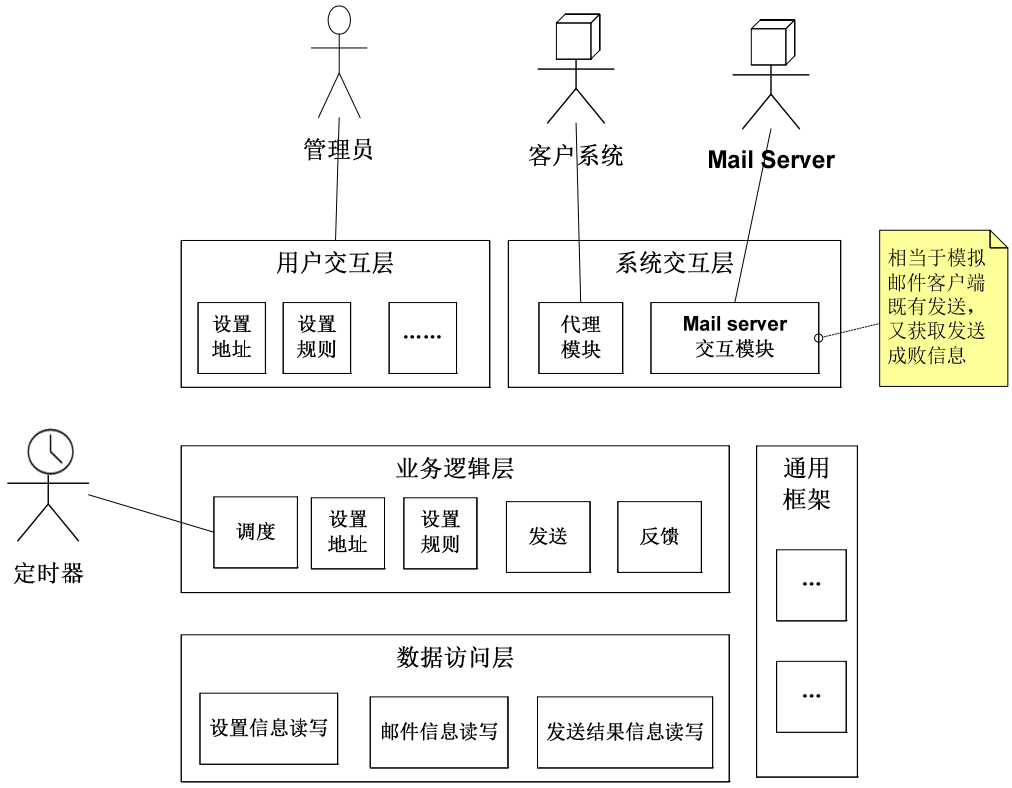


图 3-13　细化逻辑架构的设计：模块划分

好多逻辑模块呀，把它们划归 3 个工程（Project），这就意味着 MailProxy 发布时将包含 3 个目标程序单元（图 3-14 描述了 3 个目标程序单元的关系）：

- 后台服务器程序
- 管理员 Web 应用
- 代理模块（相当于 MailProxy 系统专门发布给合作开发单位的 API。这样，客户系统只需改动最少的代码就可以与 MailProxy 互联互通。运行时，客户系统必须包含代理模块的相应 Library 文件。）

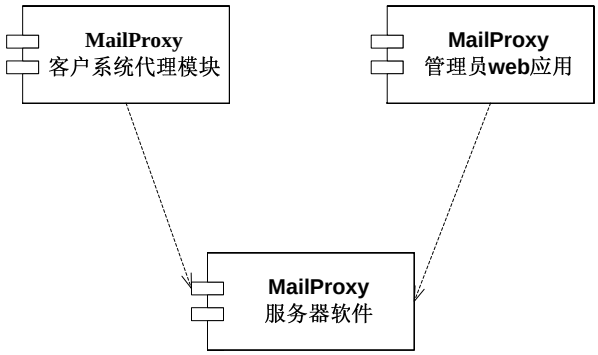


图 3-14　细化逻辑架构的设计：明确 MailProxy 包含的程序单元

3.4.6　物理架构设计（迭代 2）

继续迭代设计。

上面的逻辑架构设计逐步清晰、不同模块不断明确、相应的目标程序组件也都清楚了，使你能进一步设计物理架构了。如图 3-15 所示，软件单元和硬件机器的映射关系进一步明确了，这就为未来的安装部署方式提供了指导。

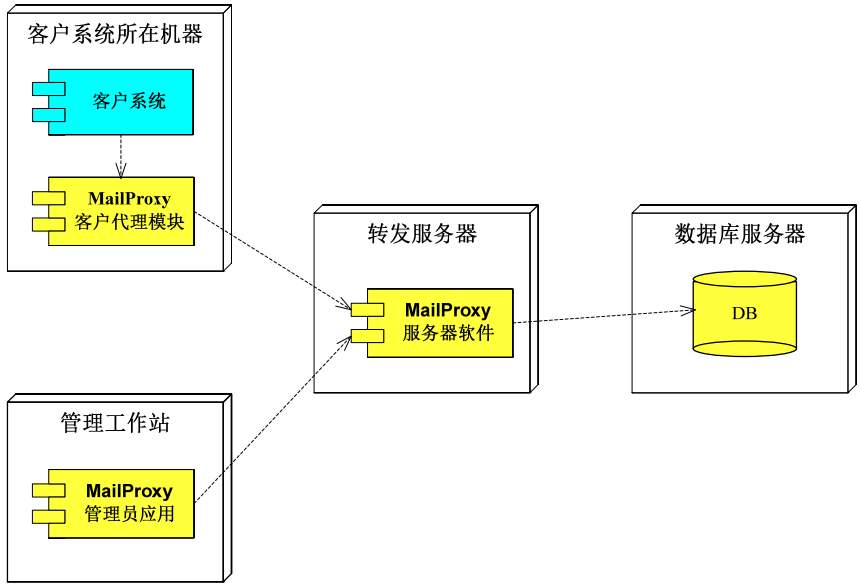


图 3-15　细化物理架构的设计：软件如何部署到硬件

嘿，多个视图之间来回迭代着进行设计，思路清晰、效果不错。

第 4 章　架构设计过程

> 作为职业软件人，我们都寻求使用一种有效而经济的过程，来建造一个能够工作的、有用的产品。
>
> ——Grady Booch，Rational 公司首席科学家
>
> 每个项目都是很独特的，因此开发人员必须努力保持微观过程的非正式性和宏观过程的正式性之间的平衡。
>
> ——Grady Booch，《面向对象分析与设计》

程序员向架构师转型，难在何处？难在必须要能开始“试着做起来”，并慢慢积累感觉、进而积累经验。

“需求决定架构”之所以是一句废话，就是因为它没告诉开发人员“架构设计怎么做”。

甚至，在没有积累任何经验和“感觉”的情况下，忽然被老板“委以重任”负责架构设计，都未必是一件好事。因为这次失败了，下次机会就没了。

本章通过下述方式，讲清楚架构设计过程的大局，希望帮助程序员能将架构设计“试着做起来”：

- 架构设计过程包含哪些步骤？
- 步骤之间什么关系？下游步骤的“输入”依赖的是上游步骤的哪个“输出”？

4.1　架构设计的实践脉络

4.1.1　洞察节奏：3 个原则

成功的架构设计是相似的，失败的架构设计各有各的原因。如下 3 个原则，可以视为做好架构设计的 3 个必要条件：

- 【原则 1】看透需求
- 【原则 2】架构大方向正确
- 【原则 3】设计好架构的各个方面

这 3 个原则，基本框定了整个架构设计过程的节奏。也就是说（如图 4-1 所示）：

- 最先，是要看透需求，这是基础。架构师可能不是“需求”和“领域模型”的负责人，但也必须深入了解。
- 中间，确定正确的概念架构。“关键需求”决定“概念架构”。
- 最后，充分设计架构的各个方面。通过多视图方法“细化架构”，通过“架构原型”验证架构。

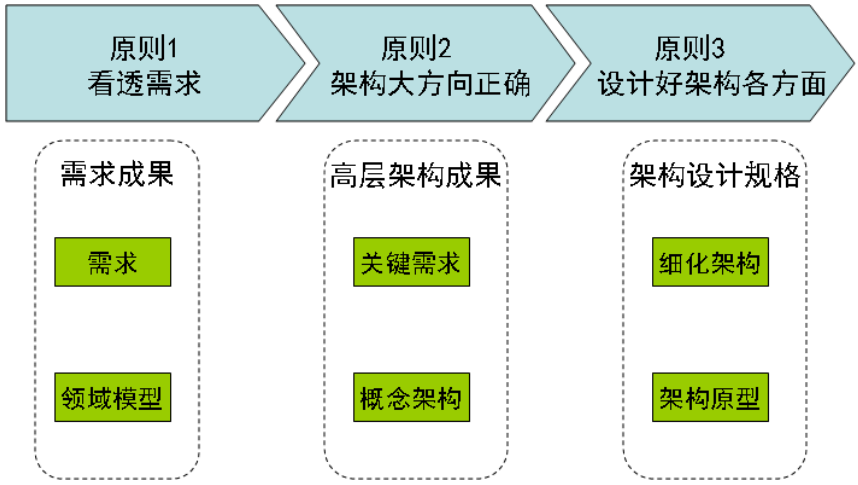


图 4-1　架构设计过程的节奏

【原则 1】看透需求

架构设计可谓影响深远，一是它决定了系统的整体质量因而决定了客户满不满意，二是它决定了开发人员能不能容易地开发、维护和扩展程序。

看透需求，简单说就是设计人员要做到“理解了、能说出所以然来”。必须的。众所周知。

看透需求，不仅要把需求找全，还要把需求项之间的矛盾关系、追溯关系也都搞清楚：

- 需求要全。

  ——举例：重视“功能”忽视“质量”，危险；重视“A 质量”忽视应有的“B 质

量”，危险；忘了来自甲方乙方第三方的“约束”，危险。功能需求、质量需求、约束需求都要定义清楚。

——处理：发现需求遗漏，就要尽快把遗漏的需求研究清楚。

- 矛盾关系。

  ——举例：安全性和互操作性有矛盾，可扩展性和性能有矛盾，功能强大和预算有限有矛盾……。

  ——处理：识别出需求间的矛盾还没完，要给对策（例如性能最重要、可扩展性折衷，或者相反首先照顾可扩展性）、甚至重新评审需求的合理性。

- 追溯关系。

  ——举例：“需求范围”合理吗？要向前追溯“系统目标”。符合目标要求、能帮助实现目标，才合理。

  ——举例：“用例图”或者“功能项定义”，是否覆盖了更高层需求之“需求范围”。

  ——处理：下层需求没有覆盖上层需求，必有需求遗漏。下层需求超出了上层需求，恐怕存在需求镀金现象。

  ——说明：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……。需求是分层次的。

【原则 2】架构大方向正确

架构大方向正确是一种策略，先设计概念架构。

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。

它不关注明确的接口定义（之后的细化架构设计才是“模块 + 接口”一级的设计），对大型系统而言，这一点恰恰是必须的。概念架构一级的设计更重视“找对路子”，像架构模式啦、集成技术选型啦……，比较策略化。

【原则 3】设计好架构的各个方面

架构师必须具备“忘却”的能力，避免涉及太多具体的技术细节，但是大型软件架构依然是复杂的。这就要求从多个方面进行架构设计，运用“多视图设计方法”：

- 例如，为了满足性能、持续可用性等方面的需求，架构师必须深入研究软件系统运行

期间的情况，权衡轻重缓急，并制定相应的并行、分时、排队、缓存和批处理等设计决策。（运行架构视图）

- 而要满足可扩展性、可重用性等方面的需求，则要求架构师深入研究软件系统开发期间的代码文件组织、变化隔离和框架使用等情况，制定相应的设计决策。（开发架构视图）
- ……本书推荐 5 视图方法。逻辑视图、开发视图、物理视图、运行视图、数据视图。

实际经验表明，越是复杂的系统，越是需要从多个方面进行架构设计，这样才能把问题研究和表达清楚。Grady Booch 在其著作《UML 用户指南》中指出：“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。”

【强调】培训中发现的问题（关于原则 2）

在培训过程中笔者发现，项目经理、架构师和总工等角色“最深有感慨”的就是“架构大方向正确”原则，但参加培训的很多程序开发人员“感触不深”。在此，笔者强调，概念架构是直指系统设计目标的设计思想和重大选择——是关乎任何复杂系统成败的最关键的、指向性的设计：

- 你作为架构师，设计大中型系统的架构时，会先对比分析几种可能的概念架构。
- 看看竞争对手的产品彩页，上面印的架构图，还是概念架构。
- 如果你是售前，你又提到架构，这也是概念架构。
- 如果你去投标，你讲的架构，就是概念架构。

架构新手和有经验架构师的区别之一，在于是否懂得、并能有效进行概念架构的设计。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 1）重大需求、2）特色需求、3）高风险需求，据此来决定如何划分顶级子系统、采用什么架构风格和开发技术、集成是否要支持、二次开发是否要支持。

4.1.2 掌握过程：6 个步骤

洞察了架构设计过程的节奏，再看步骤。整个架构设计过程，包含 6 大步骤（如图 4-2 所示）：

1) 需求分析
2) 领域建模
3) 确定关键需求
4) 概念架构设计
5) 细化架构设计
6) 架构验证

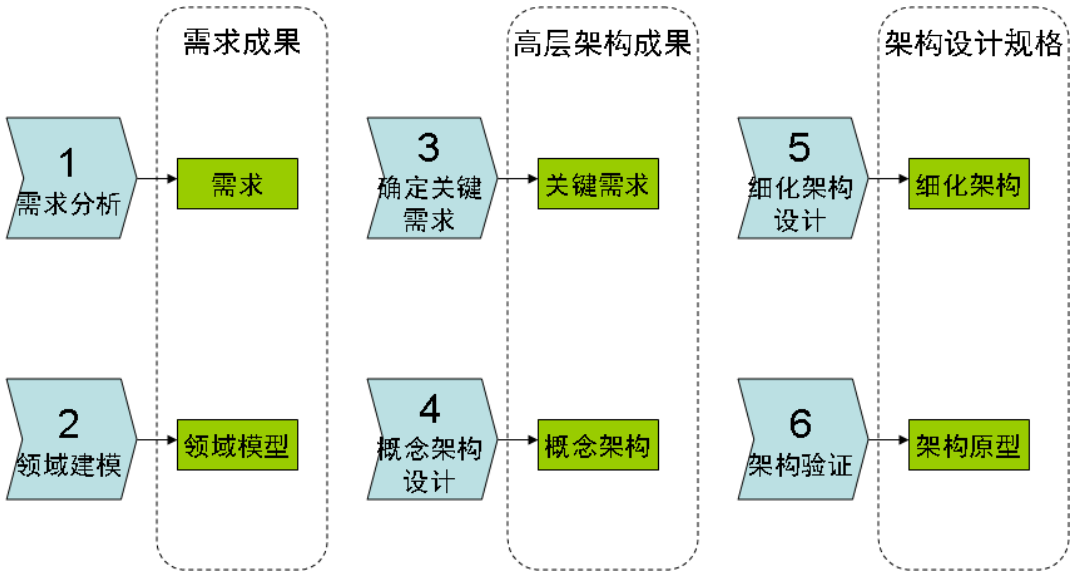


图 4-2 架构设计过程的 6 个步骤

至此，6 步骤之间的关系也就不难理解了。如图 4-3 所示。

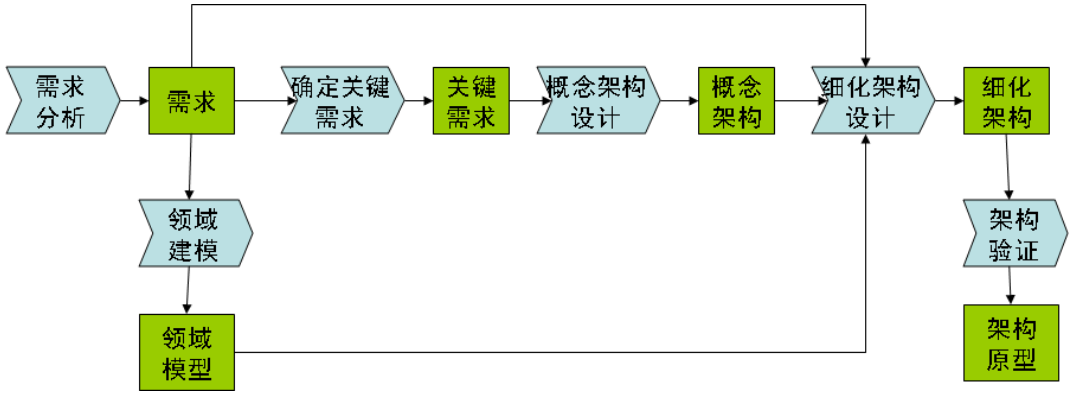


图 4-3　6 个步骤之间的关系

可以看出，架构设计的开展非常依赖其上游活动。总体而言，这些上游活动包括需求分析和领域建模。

需求分析。毋庸置疑，在没有全面认识需求并权衡不同需求之间相互影响的情况下，设计出的架构很可能有问题。

领域建模。领域建模的目的是：透过问题领域的重重现象，捕捉其背后最为稳固的领域概念及这些概念之间的关系。在项目前期，所建立的领域模型将为所有团队成员之间、团队成员和客户之间的交流提供共同认可的语言核心。随着项目的进展，领域模型不断被精化，最终成为整个软件的问题领域层，该层决定了软件系统能力的范围。本书还认为，从项目前期伊始，软件架构师就应该是领域建模活动的领导者，这样可以避免“不同阶段领域模型由不同人负责”所带来的问题。

接下来要进行概念架构的设计。软件系统的规模越大、复杂程度越高，进行概念架构设计的好处就越明显。

确定对架构关键的需求。这不仅要求对功能需求（如用例）进行筛选，还要对非功能需求进行综合权衡，最终确定对软件架构起关键作用的需求子集。

概念架构设计。概念架构的设计，必须同时重视关键功能和关键质量。业界流行的一种错误观点是“概念架构=理想化架构”，不考虑任何非功能需求，也不考虑任何具体技术。本书提出，概念架构要明确给出“1 个决定 4 个选型”，即决定：1）如何划分顶级子系统、2）架构风格选型、3）开发技术选型、4）集成技术选型、5）二次开发技术选型。可以看出，其中涉及多项重大技术选型。

在接下来，全面展开规格级的架构设计工作，设计出能实际指导团队并行开发的细化架构。

细化架构设计。一般而言，可以分别从逻辑架构、开发架构、运行架构、物理架构、数据架构等不同架构视图进行设计。

架构验证。对后续工作产生重大影响返工代价很高的任何工作都应该进行验证，软件需求如此，架构设计方案也是如此。至于验证架构的手段，对软件项目而言，往往需要开发出架构原型，并对原型进行测试和评审来达到；而对软件产品而言，可以开发一个框架（Framework）来贯彻架构设计方案，再通过在框架之上开发特定的垂直原型来验证特定的功能或质量属性。因此，从架构验证工作得到的不应该仅仅是“软件架构是否有效”的回答，还必须有可实际运行的程序：体现软件架构的垂直抛弃原型或垂直演进原型，或者是更利于重用的框架。这些成果为后续的开发提供了实在的支持。

在通过后续各章展开讨论这些工作步骤之前，再总体归纳一下每个工作步骤的主要输入、输出、关键技能项：

4.2 速查手册

下面对架构设计的每个步骤，进行总括描述，供读者在“付诸实践”时参考。

笔者建议，在系统地学习了第 5-15 章之后，请复习本节。本节所采用的这种形式，也是笔者在架构培训的“总结回顾”环节所经常采用的。效果不错。

4.2.1 需求分析

需求分析，是很多活动的统称，它是本书“架构设计过程”中第 1 个大的工作步骤。如图 4-4 所示。

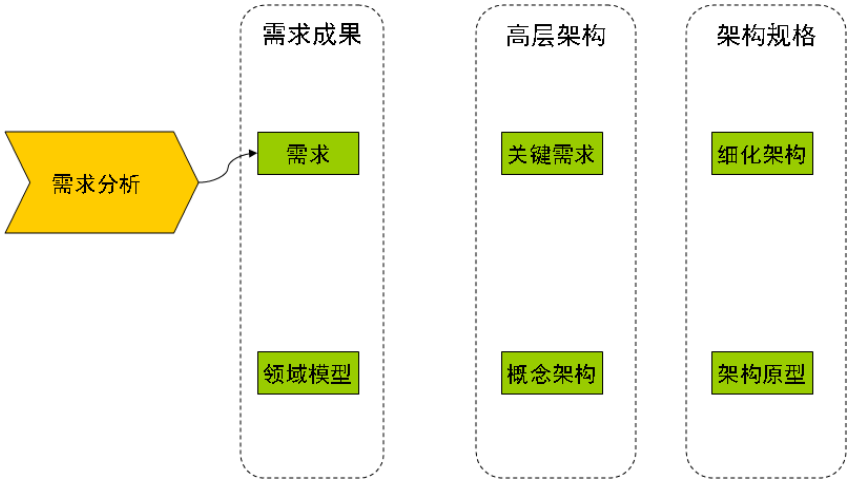


图 4-4　需求分析的“位置”

需求分析活动输出的“需求”，必须涵盖功能、质量、约束这三个方面，这些是后续设计活动所需要的。需求分析工作涉及的“技能项”较多，总体而言可总结为“两纵三横”，如图 4-5 所示：

- 【纵】需求沟通。持续伴随需求分析过程的，是需求沟通、需求启发、需求验证等活动，这些活动都要求需方和开发方紧密协同、精诚合作。“闭门造需”危险大了。
- 【纵】非功能需求的确定。真实的实践中，确定非功能需求是一个持续的过程，是持久战。究其原因，这是非功能需求的范围广造成的，无论是技术还是业务、无论是甲方还是乙方，都可能有这样那样的非功能需求。想“一蹴而就”地定义非功能需求是不现实的。
- 【三横】需求分析主线。从确定系统目标开始，后续凭借“范围+Feature+上下文图”三剑客研究高层需求，再后续建立开发人员较熟悉的用例模型。

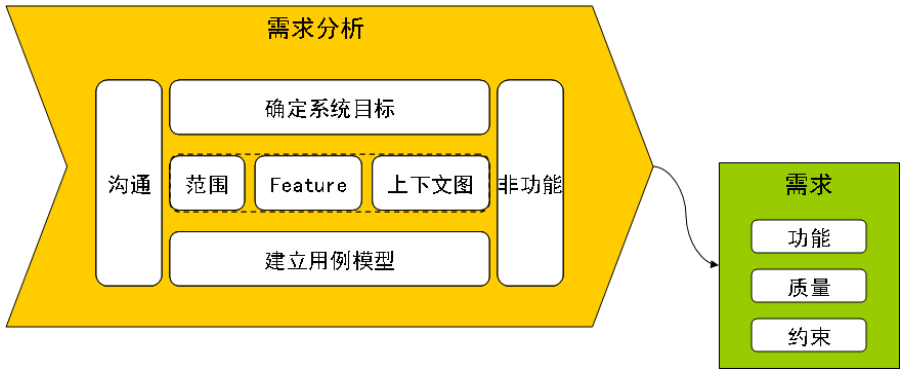


图 4-5　需求分析的“技能项”和“输出”

做不到“追根溯源”的需求分析，往往会失败。因此，我们补充图 4-6 来强调需求分析工作的主线是“确定系统目标→研究高层需求→建立用例模型”，需求从“高飘”到“落地”，成果项从“目标列表”到“范围框图 + Feature 树 + 上下文图”到“用例图 + 用例规约”，需求跟踪脉络清晰可辨。

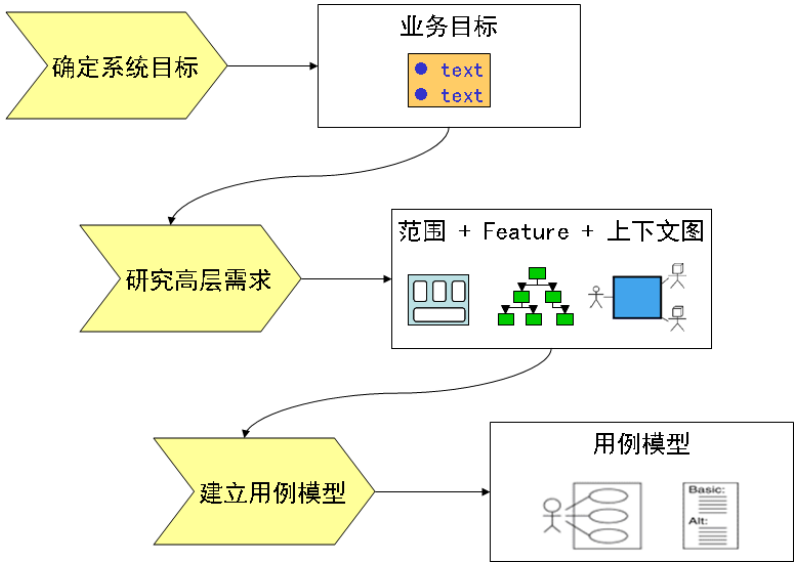


图 4-6　需求分析的主线体现“追根溯源”

更多内容，请阅读本书讲需求分析的章：第 5 章，需求分析；第 6 章，用例与需求。

4.2.2　领域建模

领域建模，是以提炼领域概念、建立领域模型为目的的活动。领域建模实践的精髓是“业务决定功能，功能决定模型”，理解了这个理念，评审领域模型也变得再自然不过了。如图 4-7 所示。

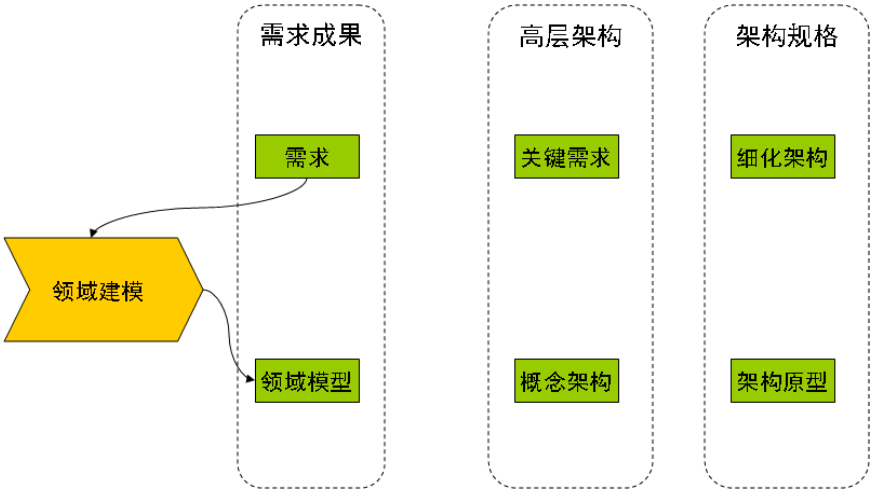


图 4-7　领域建模的“位置”

领域建模活动的输入一是“功能”，二是“可扩展性”具体要求（如图 4-8 所示）。说到底，都是“功能”，因为领域模型必须能支持在《软件需求规格说明书》中规定的“现在的功能”，还应该支持随着业务发展而出现的“未来的功能”。这两种功能，就是驱动领域建模的因素、以及评审领域模型的依据。

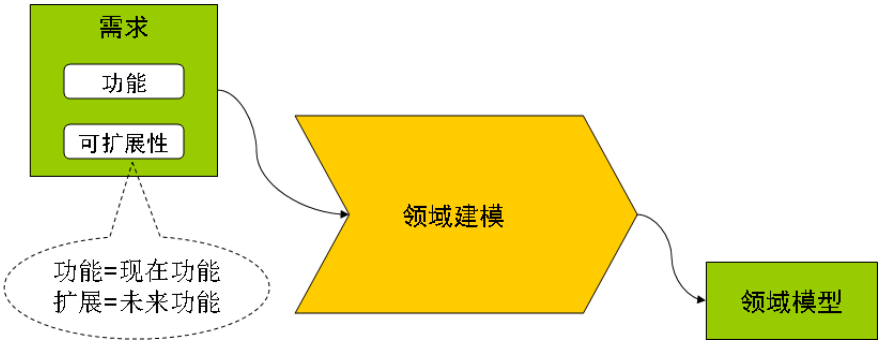


图 4-8　领域建模的“输入”和“输出”

更多内容，请阅读本书讲领域建模的章：第 7 章，领域建模。

4.2.3 确定关键需求

架构面前所有需求一律平等？不可能。

关键需求决定了架构的大方向。图 4-9 显示了确定关键需求工作的位置。

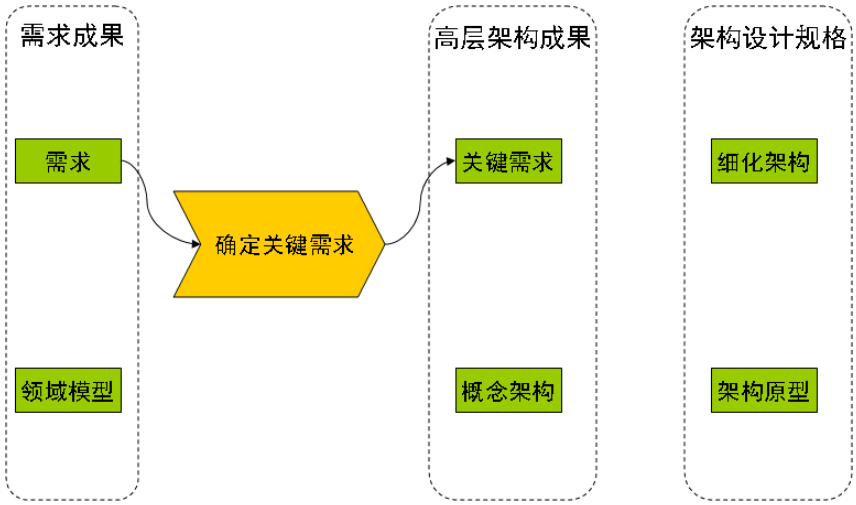


图 4-9　确定关键需求的“位置”

具体而言，为了确定“关键功能”，一要关注“功能需求”、二要研究“约束需求”；为了确定“关键质量”，一要关注“质量需求”、二也要研究“约束需求”。如图 4-10 所示。

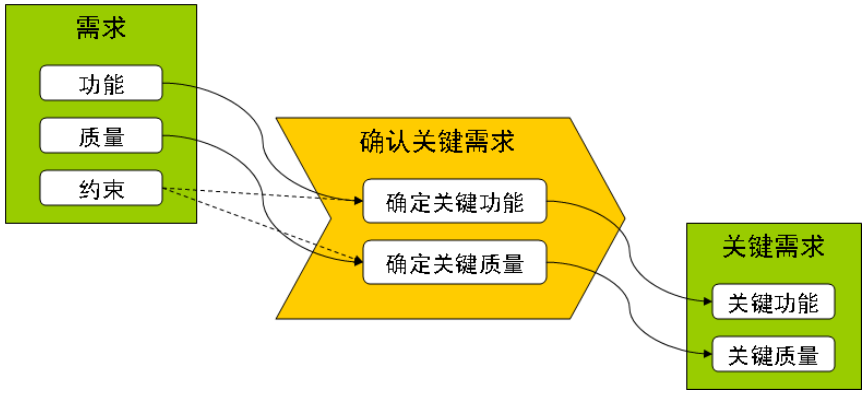


图 4-10　确定关键需求的“输入”、“技能项”和“输出”

大系统架构与小系统架构的设计为什么不同？你的系统架构与别人的系统架构设计为什么不

同？……探究架构设计的整个过程，“关键需求的确定”这一步是“分水岭”。

更多内容，请阅读本书讲确定关键需求的章：第 8 章，确定关键需求。

4.2.4 概念架构设计

概念架构是高层架构成果的核心，框定了架构大方向，是甲方规划、乙方投标的评定关键。如图 4-11 所示。

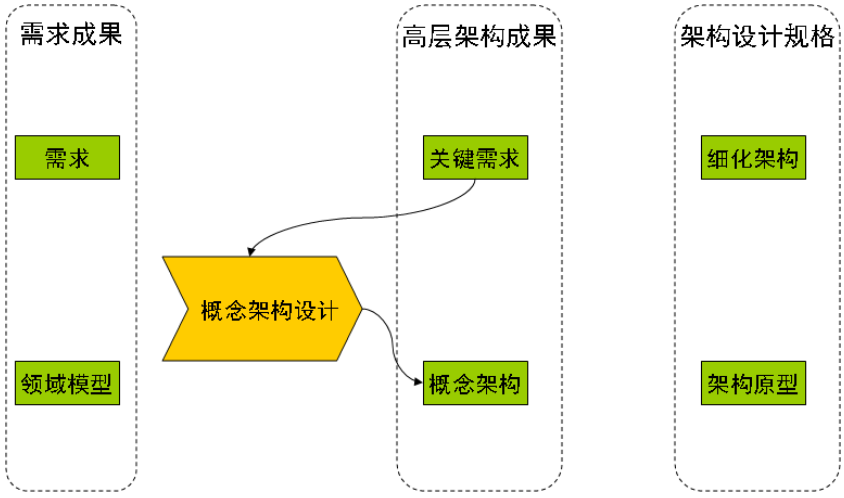


图 4-11　概念架构设计的“位置”

本书给出的定义，所谓概念架构，是直指系统目标的设计思想、重大选择。“直指目标”说的是输入，“设计思想和重大选择”说的是输出。如图 4-12 所示：

- 概念架构设计要“直指”的、以之为输入的，就是“关键需求”。
- 针对不同需求（功能 OR 质量），需要运用不同“技能项”，鲁棒图建模、目标-场景-决策表，非常实用。
- 概念架构设计的“输出”是“1 个决定、4 个选择”：
  1) 决定如何划分顶级子系统；
  2) 架构风格选型；
  3) 开发技术选型；
  4) 二次开发技术选型；

5)　集成技术选型。

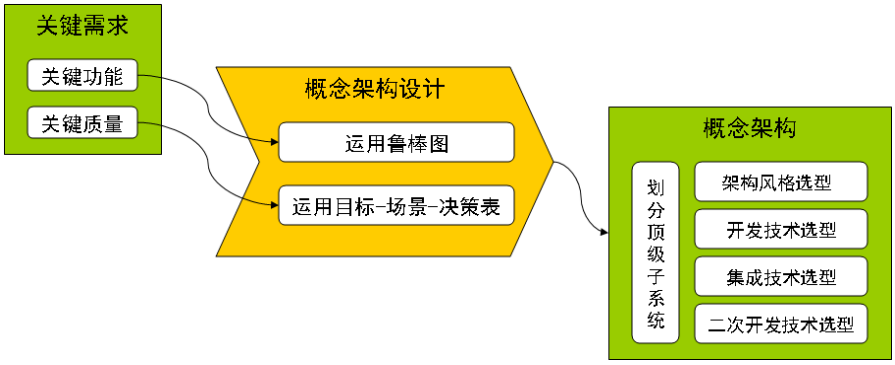


图 4-12　概念架构设计的“输入”、“技能项”和“输出”

更多内容，请阅读本书讲概念架构设计的章：第 9 章，概念架构设计。

4.2.5　细化架构设计

细化架构和概念架构的关键区别之一是：概念架构没有设计到“模块 + 接口”一级，而细化架构必须关注“模块 + 接口”。图 4-13 所示为细化架构设计的“位置”。

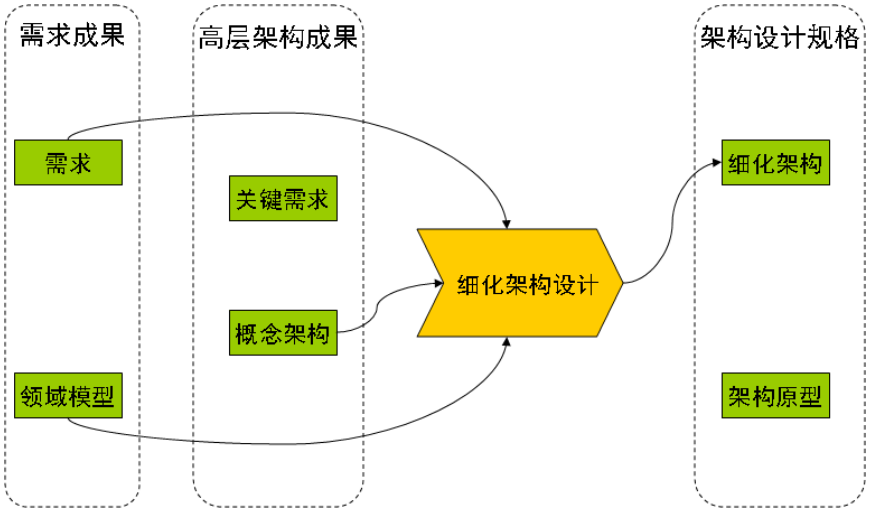
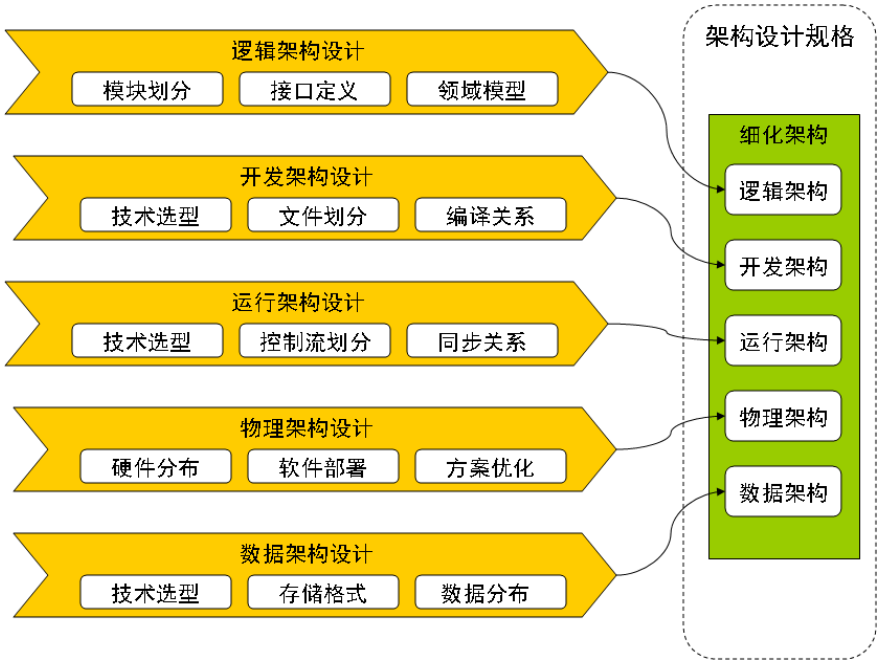


图 4-13　细化架构设计的“位置”

众所周知，架构设计涉及的方面很广（模块切分呀、持久化格式呀、并行并发呀都得管），架构设计师得是通才（要掌握的技能项较多）。对此，细化架构设计这一步体现得最充分，图 4-14 列出了细化架构设计的 5 个设计视图、15 个设计任务。



题 4-14　细化架构设计的“技能项”——15 个设计任务

再关注上游对细化架构设计的影响、支持。细化架构设计的输入既来自“需求成果”层面、也来自“高层架构”层面，如图 4-15 所示：

- 细化架构要为“需求”而设计。关键对比：概念架构设计的输入是“关键需求”、而不是泛泛的所有“需求”。
- 细化架构要在“概念架构”的设计思想下进行。
- “领域模型”，一方面影响着“逻辑架构视图”的“领域模型设计”，另一方面影响着“数据架构视图”的“存储格式设计”。

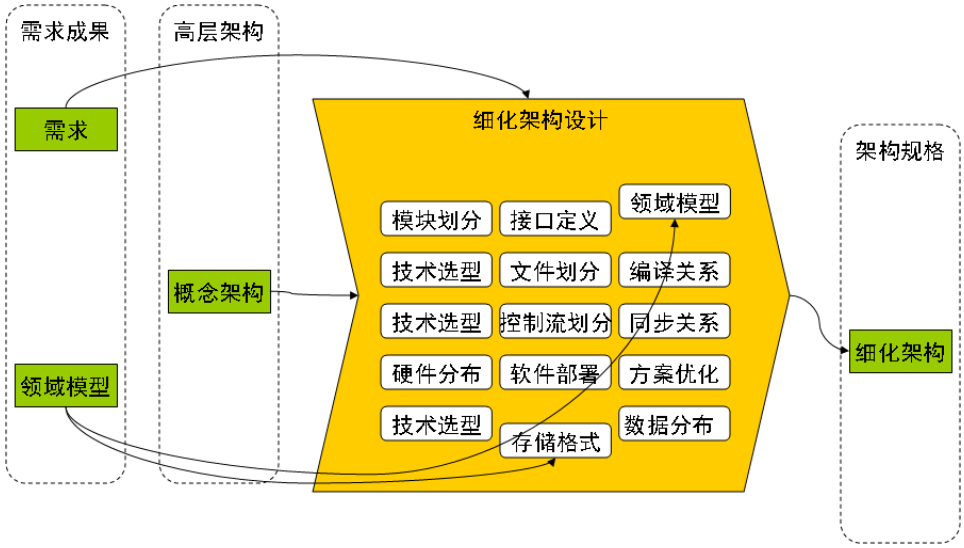


图 4-15　细化架构设计的“输入”和“输出”

更多内容，请阅读本书讲细化架构设计的章：

- 第 10 章，细化架构设计。
- 第 12 章，粗粒度“功能模块”划分。
- 第 13 章，如何分层。
- 第 14 章，用例驱动的模块划分过程。
- 第 15 章，模块划分的 4 步骤方法——运用层、模块、功能模块、用例驱动。

4.2.6　架构验证

如有必要，需要进行架构验证。如图 4-16 所示。

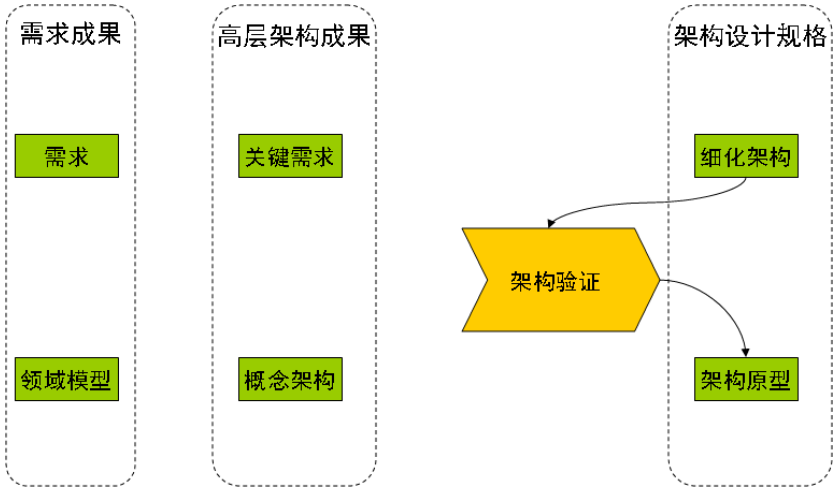


图 4-16　架构验证的“位置”

架构验证的输出成果是“架构原型”。和一般的开发不同，架构原型的开发不是要完美地、无 Bug 地实现功能，而是在“细化架构”的总体指导下、仅把存在“风险”的那些设计尽早开发出来，然后通过执行测试等手段判断“风险”是否解决。如图 4-17 所示。

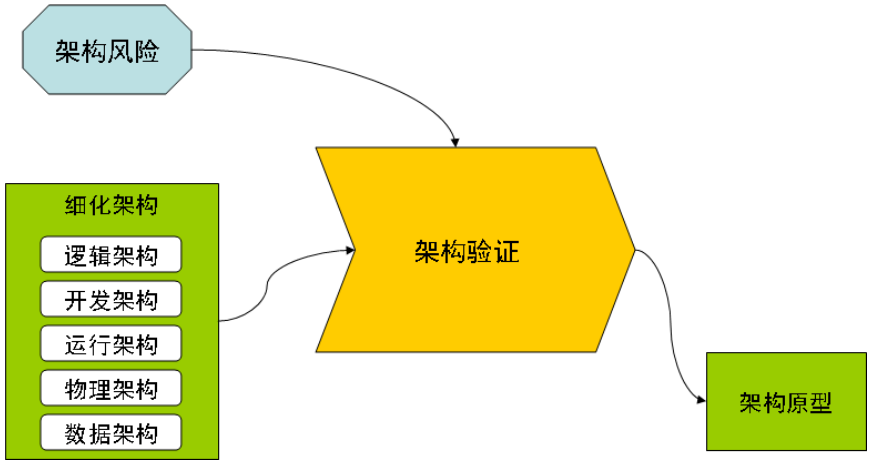


图 4-17　架构验证的“输入”和“输出”

更多内容，请阅读本书讲架构验证的章：第 11 章，架构验证。

第 9 章　概念架构设计

> 概念架构是一个“架构设计阶段”，……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。
>
> ——温昱，《一线架构师实践指南》

> 架构被用作销售手段，而不是技术蓝图，这屡见不鲜（Too often, architectures are used as sales tools rather than technical blueprints.）。
>
> ——Thomas J Mowbray，《What is Architecture》

概念架构是直指系统目标的设计思想、重大选择，因而非常重要。《方案建议书》、《技术白皮书》和市场彩页中，都有它的身影，以说明产品/项目/方案的技术优势。也因此，有人称它为“市场架构”。

大量软件企业，招聘系统架构师（SA）、系统工程师（SE）、技术经理、售前技术顾问、方案经理时，职位能力中其实都包含了对“概念架构设计能力”的要求。例如：

- 系统架构师（SA）。1、软件总体设计、开发及相关设计文档编写；2、关键技术和算法设计研究；3、系统及技术解决方案设计，软件总体架构的搭建；4、通讯协议设计制定、跟踪研究；……
- System Architect（SA）。Participates in client communications to understand client's business requirement and propose alternative solutions……
- 系统工程师（SE）。产品需求分析；产品系统设计；技术问题攻关；解决方案的输出和重点客户引导；指导开发工程师对产品需求进行开发……
- 技术经理。负责公司系统的架构设计，承担从业务向技术转换的桥梁作用；协助项目经理制定项目计划和项目进度控制；辅助需求分析师开展需求分析、需求文档编写工作；……
- 售前技术顾问。1、负责支持大客户解决方案和能力售前咨询工作；2、完成项目售前阶段的客户调研、需求分析和方案制定、协调交付部门完成 POC 或 Demo；3、参与答标，负责标书澄清；4、参与项目项目前期或高层架构设计，根据需要完成项目的系统

设计相关工作；……

- 解决方案经理。解决方案提炼与推广；现场售前技术支持，如市场策划、方案编写，售前交流等；为前端市场人员提供投标支持、投标方案（技术、配置）编制或审核；……

既然概念架构这么重要，本章就专门讲述：

- 概念架构“是什么”？
- 概念架构“长什么样”？（案例分析）
- 概念架构“怎么设计”？

9.1 什么是概念架构

9.1.1 概念架构是直指目标的设计思想、重大选择

概念架构，英文是 Conceptual Architecture。至于概念架构的定义，Dana Bredemeyer 等专家是这么阐释的：

> 概念架构界定系统的高层组件、以及它们之间的关系。概念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。概念架构规定了每个组件的非正式规约、以及架构图，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.)

根据上述定义，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）。

- 而且，必须地，概念架构不应涉及接口细节（without interface detail）。

上述定义从实践来看并不令人满意。讲课时，笔者这样给概念架构下定义：

概念架构是直指目标的设计思想、重大选择。

结合案例来理解：

9.1.2 案例 1：汽车电子 AUTOSAR——跨平台复用

嵌入式系统的应用覆盖航空航天、轨道交通、汽车电子、消费电子、网络通讯、数字家电、工业控制、仪器仪表、智能 IC 卡、国防军事等众多领域。

中国汽车产业及市场受到全球半导体产业的关注，中国汽车电子市场飞速发展已成趋势。2009 年，中国成为世界第一汽车生产大国。2010 年，中国汽车产销超过 1800 万辆。汽车电子嵌入式软件架构的标准化是汽车行业不可阻挡的发展趋势，目前比较著名的标准有 OSEK/VDX 和 AUTOSAR 标准等。

AUTOSAR 即 AUTomotive Open System Architecture（汽车开放系统架构），旨在推动建立汽车电气/电子(E/E)架构的开放式标准，使其成为汽车嵌入式应用功能管理的基础架构，如图 9-1 所示。该组织的会员企业横跨汽车、电子和软件等行业，包括宝马(BMW)、博世(Bosch)、Continental、戴姆勒克莱斯勒、福特、通用汽车、标致雪铁龙(PSA)、西门子 VDO、丰田和大众(Volkswagen)。

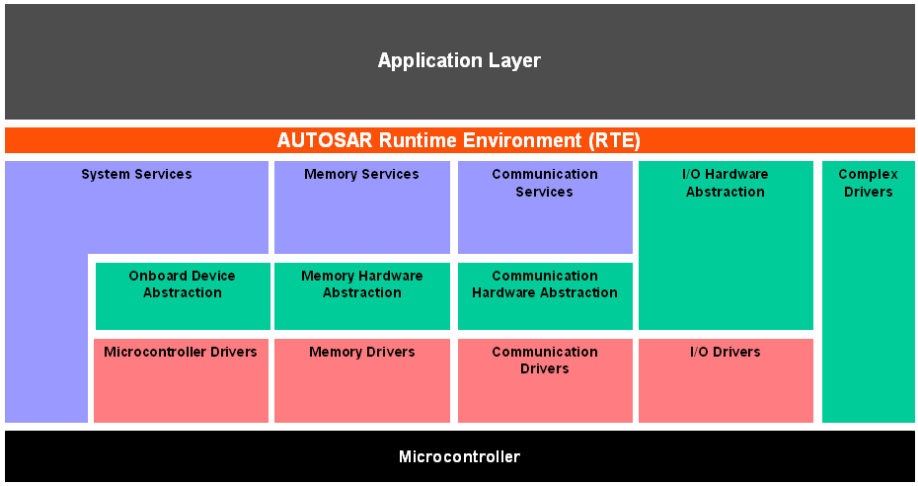


图 9-1　AUTOSAR 软件架构层次图

先说“直指目标”。

汽车电子是车体电子控制装置和车载电子控制装置的总称。车体汽车电子控制装置，包括发动机控制系统、底盘控制系统和车身电子控制系统（车身电子 ECU）。用传感器、微处理器 MPU、执行器、数十甚至上百个电子元器件及其零部件组成的电控系统，不断提高着汽车的安全性、舒适性、经济性和娱乐性。

AUTOSAR 必须应对，当前汽车电子系统越来越复杂的趋势。以中级车为例，国内的中级车大概是 10~20 个 ECU，国外的可能超过了 40 个。顶级车型的 ECU 数量甚至超过 70 个。汽车产业链中的众多厂商日益发现，兼容性差、重用性差的问题已经影响到企业利益乃至整个行业的技术创新……。其实，兼容性差、重用性差，也给用户带来不便，《中国电子报》上就讲了这么一件事儿：

> 同事老王前段时间很郁闷，他买的车故障诊断器坏了，需要换一个，可他跑了几家 4S 店，都没有他这一型号的故障诊断器。4S 店的工作人员告诉他，因为故障诊断器有多家供应商，他们不可能把所有的故障诊断器都进货，而且故障诊断器要与汽车电子控制单元中基础软件当中的故障诊断程序相通，虽然他们也与供应商谈过，不过他们都不愿统一，这需要整车厂去协调。

简要提炼出“兼容性差、重用性差”产生的大背景：

- 一方面，现代汽车电子系统从单一控制发展到多变量多任务协调控制、软件越来越庞大、越来越复杂；
- 另一方面，汽车功能创新不断、多样化差异化加剧，给汽车电子系统的研发提出更多创新、定制、快速上市的研发要求。

在这样的大背景下，AUTOSAR 架构“直指”的目标就是：

能够跨平台、跨产品复用软件模块；
避免不同产品之间的代码复制、反复开发和版本增殖问题。

再看“设计思想”。

为了实现跨平台复用的目标，AUTOSAR 采用的关键设计思想是什么呢？

关键设计思想之一是应用层、服务层、ECU 抽象层、微控制器抽象层的分离，如图 9-2 所示。这样一来，控制器硬件相关部分、ECU 硬件相关部分、硬件无关部分被分开了，利于重用。从下层到上层：

- 微控制器抽象层（Microcontroller Abstraction Layer）包括与微控制器相关的驱动，封装微控制器的控制细节，使得上层模块独立于微控制器。
- ECU 抽象层（ECU Abstraction Layer）将 ECU 结构进行抽象、封装。该层的实现与 ECU 硬件相关，但与控制器无关。
- 服务层（Services Layer）提供包括网络服务、存储服务、操作系统服务、汽车网络通讯和管理服务、诊断服务和 ECU 状态管理等相关的系统服务。除操作系统外，服务层的软件模块都是与平台无关的。
- 应用层（Application Layer）包括各种应用程序，与硬件无关。

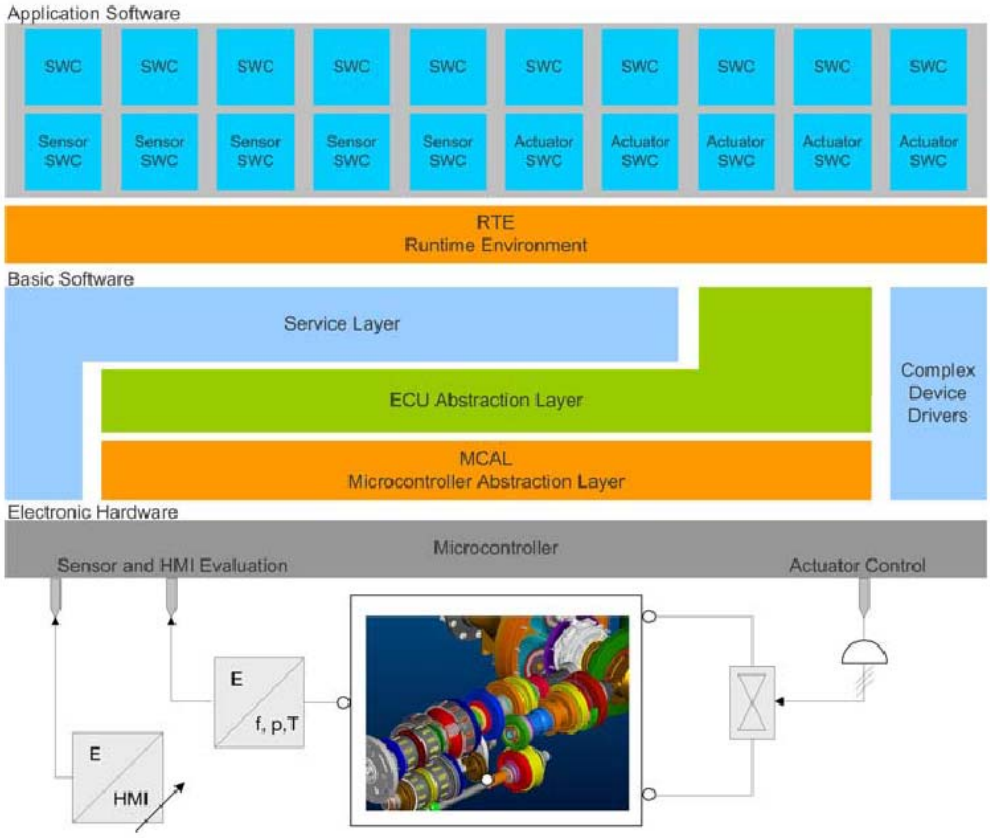


图 9-2　AUTOSAR 软件架构层次图

关键设计思想之二 RTE。

如果没有 RTE（Runtime Environment），不同软件构件的创建、销毁、服务调用、数据传递等处理都由构件自身直接负责，不仅增加了复杂性，还使构件之间相互耦合、难以灵活替换和重用。例如，构件 A 需要调用构件 B、C、D、E、F，如果要求构件 A 硬编码来创建 B、C、D、E、F 就太烦了，而且在每个构件都可能调用一堆其他构件时这种“直接创建和销毁”的方式显然是不可行的。

如图 9-3 所示，AUTOSAR 采用了构件化思想，RTE 的本质就是构件运行环境。具体而言，RTE 负责 3 件事：

- 构件生命周期管理（上面讨论的构件的创建、销毁问题有“人”管了）
- 构件运行管理
- 构件通信管理

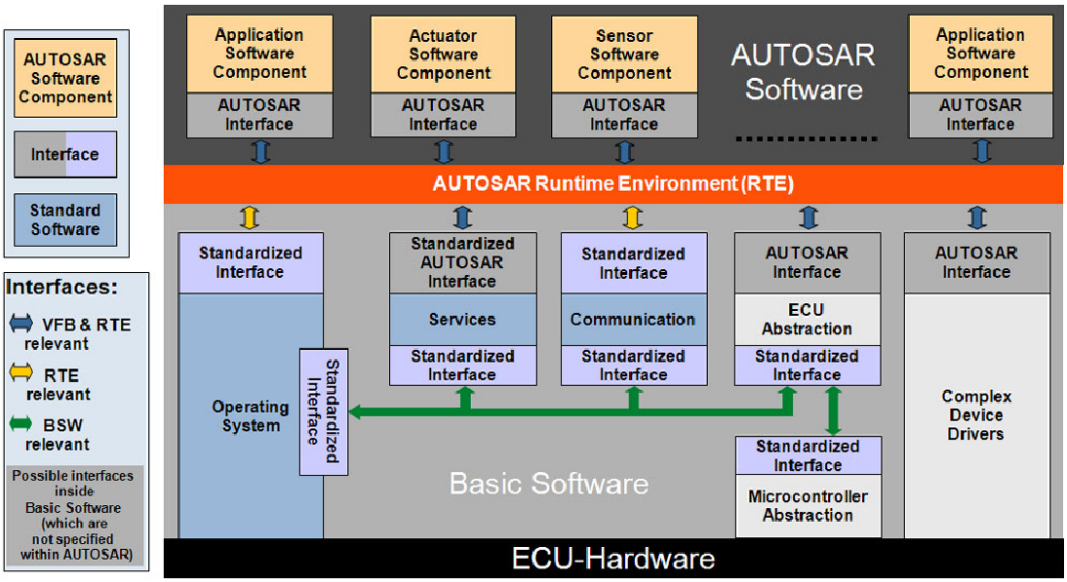


图 9-3　RTE 是 AUTOSAR 的关键设计

至此，RTE 层之下的基础软件对于应用层来说就不可见了。实际上，AUTOSAR 还提供相关接口的标准定义、元数据配置等手段，进一步支撑起“在标准上合作，在实现上竞争”的产业链原则。

AUTOSAR 架构包含（和隐含）了哪些“重大选择”呢？

在集中式、分布式上，AUTOSAR 选择了分布式架构。

集中式架构所有车身电器由一个控制器控制，缺点明显：

- 控制状态复杂、容易出错、可靠性差
- 重用性差、新产品开发周期长
- 不同产品之间的代码复制、反复开发和版本增殖问题

分布式架构通过各个子系统交互来控制整车（如图 9-4 所示），优点有：

- 配置灵活、扩展方便、易于支持各种丰富功能

■ 单个子系统可靠性高、重用性好

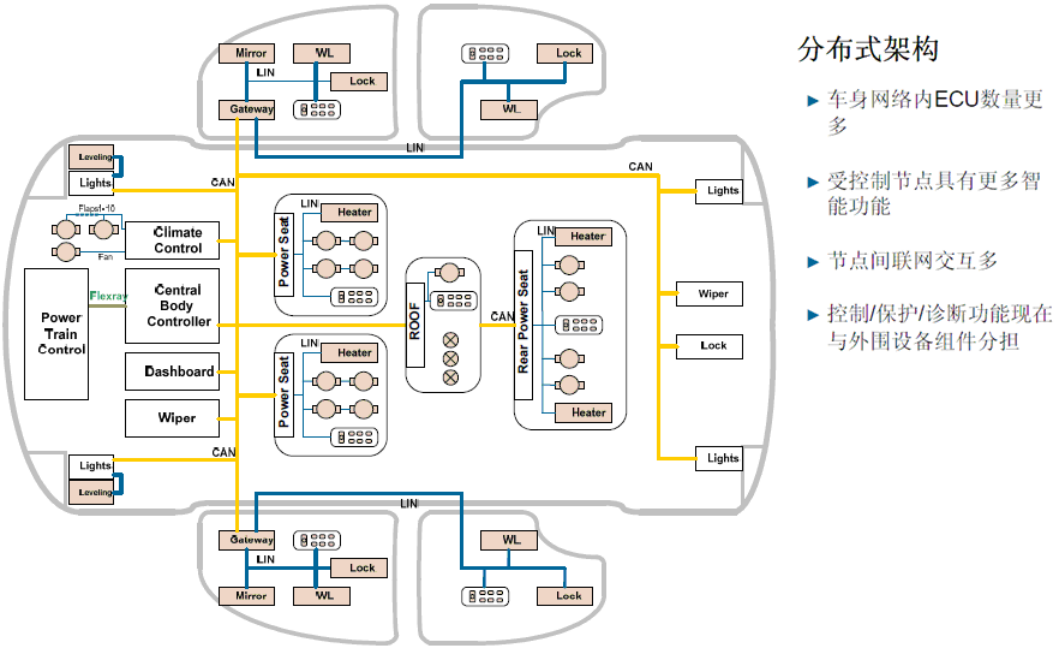


图 9-4　汽车电子的分布式架构（来源：飞思卡尔技术资料）

AUTOSAR 架构中的另一个选择也很有意义——复杂设备驱动（如图 9-5 所示）。

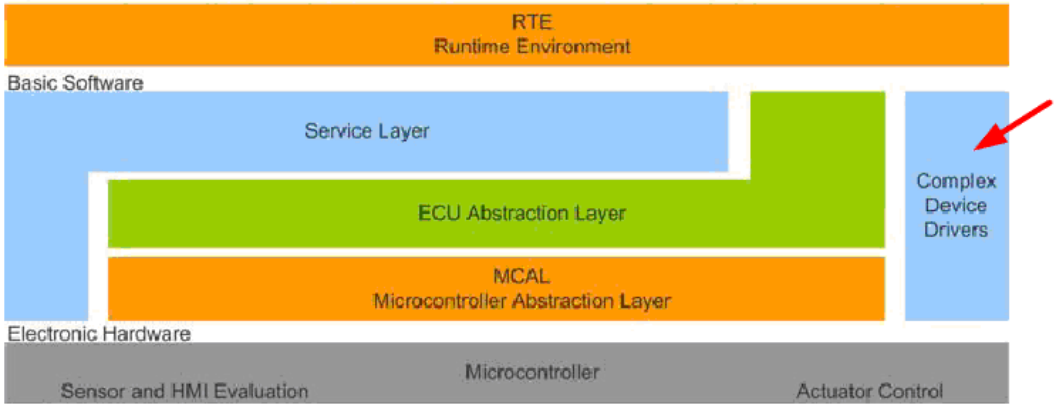


图 9-5　复杂设备驱动：放弃“层次化”，选择“一体化”

复杂设备驱动（(Complex Device Driver, CDD)）的设计选择的是“一体化”甚至“一锅粥”的设计，而没有采用“服务层—ECU 抽象层—微控制器抽象层”的“层次化”设计。显然，这种设计明显带有折衷的性质，但相当明智！究其原因，在 AUTOSAR 标准中复杂设备驱动模块可以直接访问 ECU 基础软件和微控制器硬件，也可以集成 AUTOSAR 标准中未定义的微控制器接口，以便处理对复杂传感器和执行器进行操作时涉及的严格时序问题。（好熟悉的设计思想！让人不由想起了微软的 DirectX 架构。）

9.1.3 案例 2：腾讯 QQvideo 架构——高性能

架构设计中充满了选择，概念架构要做重大选择。

如果你是一个 Web 系统的架构师，你会选择“水平分层 + 统一管理各类资源”这种设计呢，还是选择“不同功能垂直分离 + 资源分别对待”这种设计呢？

腾讯 QQvideo 架构，选择的是后者，如图 9-6 所示：

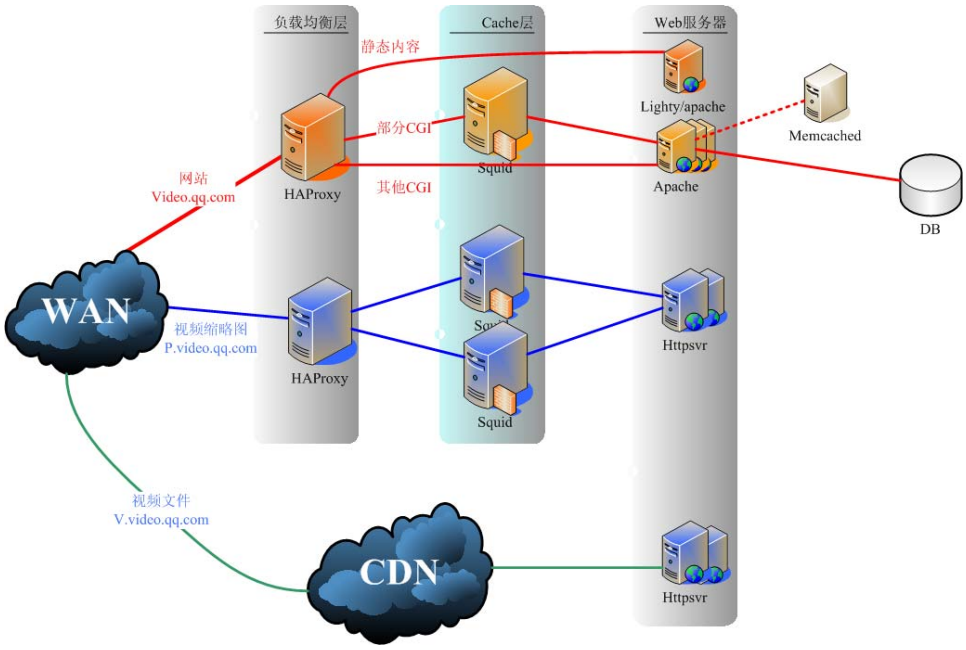


图 9-6　QQvideo Web 架构图（图片来源：腾讯大讲堂 http://djt.open.qq.com）

既然是概念架构举例，我们就扣扣定义（概念架构是直指系统目标的设计思想、重大选择）：

- 直指目标：高性能。（以及和高性能密切相关的可伸缩性）

- 设计思想：不同功能垂直分离
  - 视频、视频缩略图、静态 Web 内容、动态 Web 内容，分别对待
- 重大选择：首先垂直划分系统、而不是水平分层

9.1.4 案例 3：微软 MFC 架构——简化开发

平台和应用，是互补品的关系。

微软一直以来都非常重视对开发者的支持，就是希望更多的个人和公司在 Windows 平台上开发应用系统。Windows 平台上的互补应用产品越丰富，就越能吸引和“拴住”用户。

MFC（Microsoft Foundation Classes）是微软较早时推出的 Application Framework，至今仍有比较广泛应用。图 9-7 所示为微软 MFC 的概念架构：

- 直指目标：提供比“使用 Win32 API”更方便的应用开发支持
- 设计思想：应用支持层 + 抽象引入层 + Win32 封装层
- 重大选择：微软放弃了此前的 AFX 设计（AFX 失败了），转而采用首先“封装 Win32 API”的策略，更务实、更有可能成功。

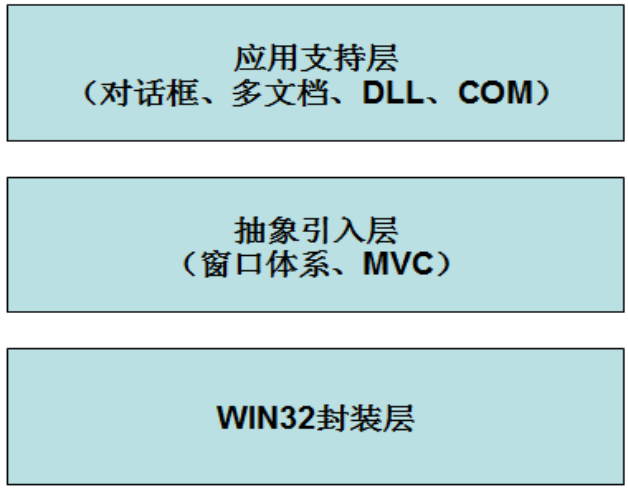


图 9-7　微软 MFC 概念架构

9.1.5 总结

架构师是设计的行家里手，有很多思想。但只有设计思想与目标联系时，才可能取得成效。

概念架构，就是直指系统设计目标的设计思想和重大选择——是关乎任何复杂系统成败的最关键的、指向性的设计。概念架构贵在有针对性，“直指目标”、“设计思想”和“重大选择”是它的三大特征。

你是否意识到，你在实际工作中已多次接触过概念架构了呢：

- 你作为架构师，设计大中型系统的架构时，会先对比分析几种可能的概念架构。
- 看看竞争对手的产品彩页，上面印的架构图，还是概念架构。
- 如果你是售前，你又提到架构，这也是概念架构。
- 如果你去投标，你讲的架构，就是概念架构。

9.2 概念架构设计概述

9.2.1 “关键需求”进，“概念架构”出

有经验的架构师知道，花精力去明确对架构设计影响重大的关键需求非常值得。本书前面几章的内容做到了这一点（如图 9-8 所示）：

- 第 5 章，讲需求分析
  第 6 章，又专门讨论了用例技术
- 第 7 章，讲领域建模
- 第 8 章，讲如何确定关键需求

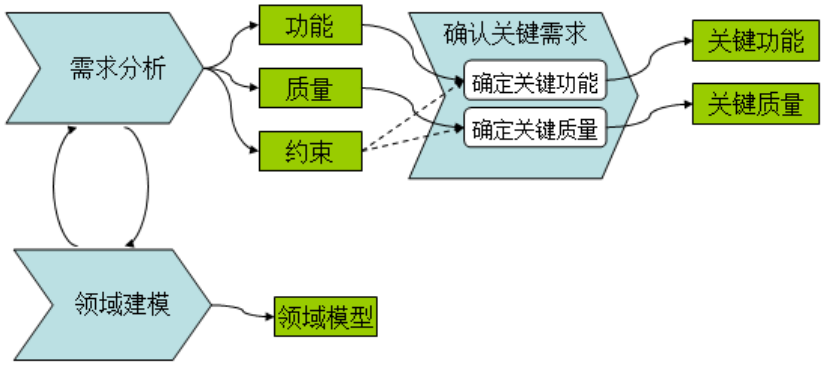


图 9-8　需求分析、领域建模和确定关键需求

明确了关键需求，接下来要设计概念架构——就是以“关键需求”为目标，明确设计思想，进行重大设计选择。概括而言，概念架构设计过程是个“关键需求进，概念架构出”的过程，如图 9-9 所示：

- 针对关键功能，运用鲁棒图进行设计；（本章第 3 节讲解）
- 针对关键质量，运用目标-场景-决策表设计。（本章第 4 节讲解）

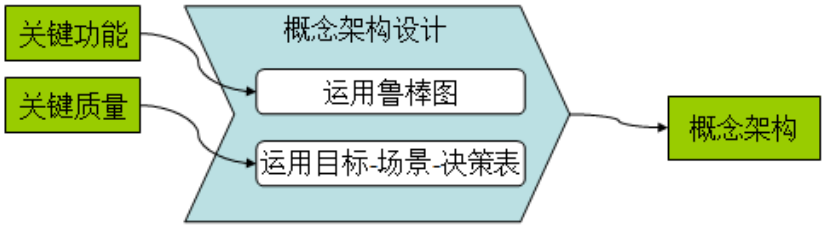


图 9-9　“关键需求”进，“概念架构”出

9.2.2　概念架构≠理想化架构

实际工作当中，单纯采用功能需求驱动的方式，未免太理想化了，会造成“概念架构 = 理想化架构”错误。

所谓理想化架构，就是一门儿心思只考虑功能需求、不考虑非功能需求而设计出来的架构。例如，总是忽略硬件限制、带宽限制、专利限制、使用环境、网络攻击、系统整合、平台兼容……等现实存在的各种约束性需求，这样设计出来的概念架构后期可能需要大改。

9.2.3 概念架构≠细化架构

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。所以，概念架构≠细化架构。

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，一个软件产品与竞争对手在架构上的不同，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，这一点恰恰是必须的。总结起来，“概念架构≠细化架构”涉及了开发人员最为关心的多项工作：

- 接口。在细化架构中，应当给出接口的明确定义；而概念架构中即使识别出了接口，也没有接口的明确定义；
- 模块。细化架构重视通过模块来分割整个系统，并且模块往往有明确的接口；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；
- 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是基于接口编程、消息机制或远程方法调用等其中的任一种。
- 因此，概念架构是不可直接实现的。开发人员拿到概念架构设计方案，依然无法开始具体的开发工作。从概念架构到细化架构，要运用很多具体的设计技术，开发出能够为具体开发提供更多指导和限制的细化架构。

9.3 左手功能——概念架构设计（上）

如前所述，概念架构设计环节的输入，一是关键功能、二是关键质量。形象地说，叫左手功能、右手质量，两手抓，两手都要硬。

9.3.1 什么样的鸿沟，架什么样的桥

需求和设计之间存在一道无形的鸿沟，因此很多人会在需求分析之后卡壳，不知道怎么做了。

先说功能需求。使用用例规约等技术描述功能，可以阐明待开发系统的使用方法，但并没有以类、包、组件、子系统等元素形式描述系统的内部结构。从用例规约向这些设计概念过渡之所以困难，是因为如下一些原因：

- 用例是面向问题域的，设计是面向机器域的，这两个“空间”之间存在映射；
- 用例技术本身不是面向对象的，而设计应该是面向对象的，这是两种不同的思维方式；
- 用例规约采用自然语言描述，而设计采用形式化的模型描述，描述手段也不同。

越过从功能需求到设计的鸿沟，需要搭桥。这“桥”就是下面要讲的鲁棒图建模技术：

9.3.2 鲁棒图“是什么”

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。

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）”。

9.3.3 鲁棒图“画什么”

鲁棒图包含 3 种元素（如图 9-10 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。

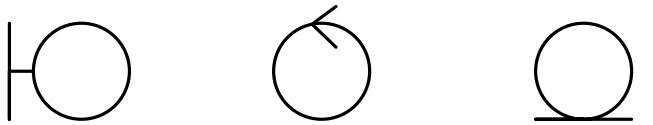


图 9-10　　鲁棒图的元素

整个系统，会涉及很多用例。每个用例（Use Case）＝N 个场景（Scenario）。每个场景（可能是正常场景、也可能是各种意外场景），其实现都是一串职责的协作。实践中，经常是从一个个用例规约等功能需求描述着手，基于鲁棒图建模技术，不断发现场景背后应该有哪些不同的职责（如图 9-11 所示）。

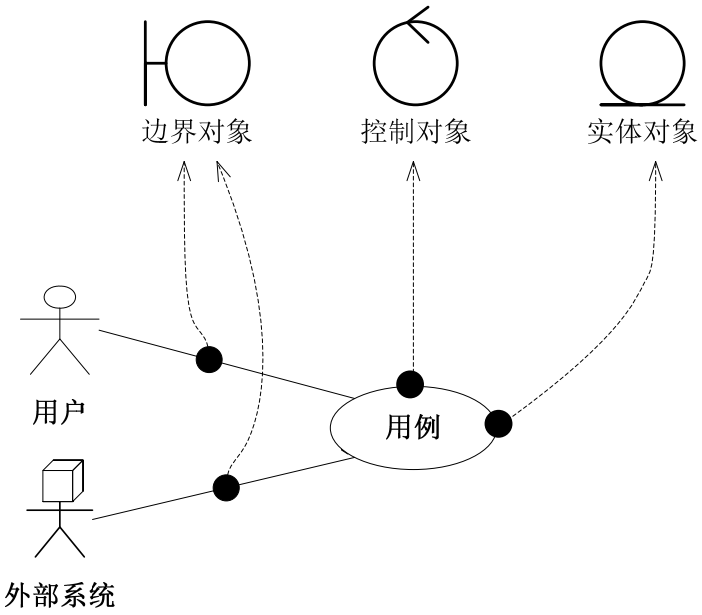


图 9-11　研究用例，进行鲁棒图建模

也就是说，研究功能如何和外部 Actor 交互而发现边界对象，研究功能实现需要的行为而发现控制对象，研究实现功能所需要的必要信息而发现实体对象。这种从功能需求到基本设计元素（交互、行为、信息）的思维过程非常直观，抹平了从需求到设计“关山难越”的问题（如图 9-12 所示）。

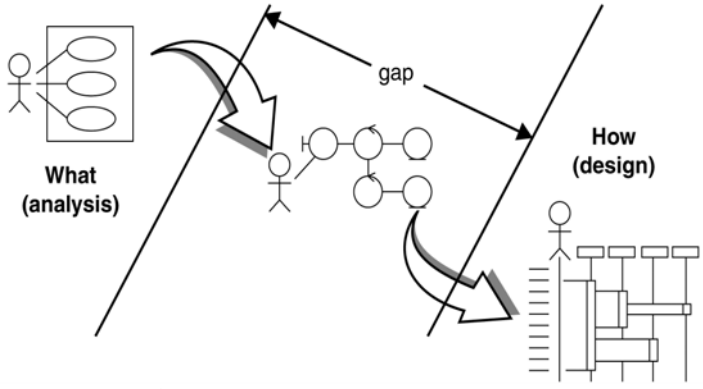


图 9-12　鲁棒图的“桥梁”作用（图片来源：《UML 用例驱动对象建模》）

图 9-13 进一步具体化了边界对象、控制对象和实体对象所能覆盖的交互、行为和信息这三

种职责，实践中可多应用多体会。另外就是要强调，鲁棒图 3 元素和 MVC 还是有不小差异的，实践中勿简单等同：

- 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

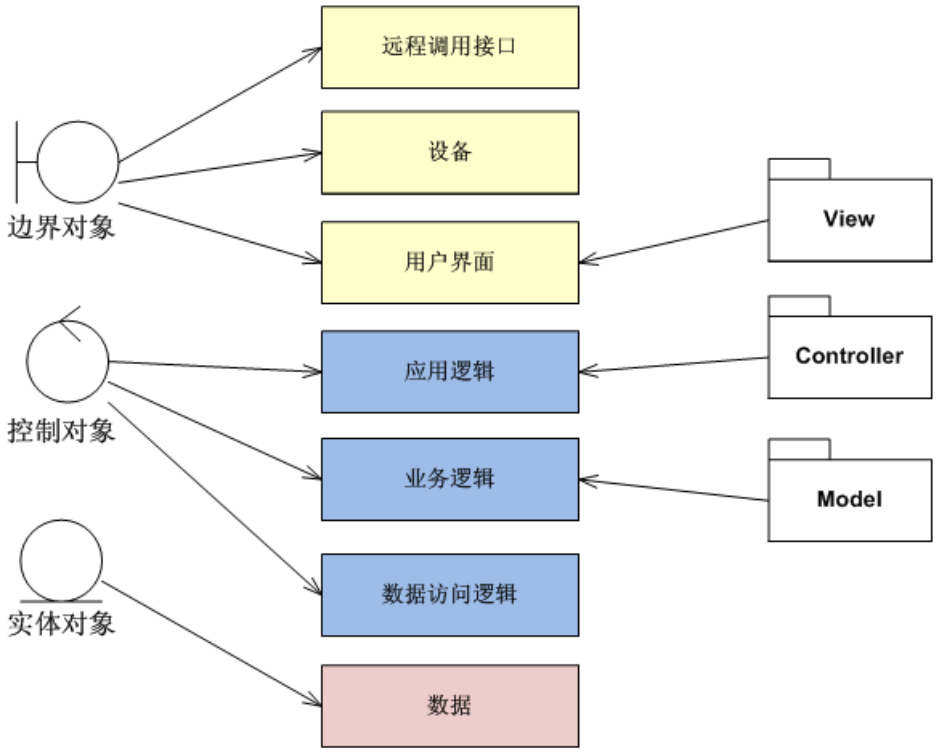


图 9-13　鲁棒图 3 元素的具体化、以及与 MVC 的对比

9.3.4 鲁棒图“怎么画”

图 9-14 所示，是银行储蓄系统的“销户”功能的鲁棒图。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

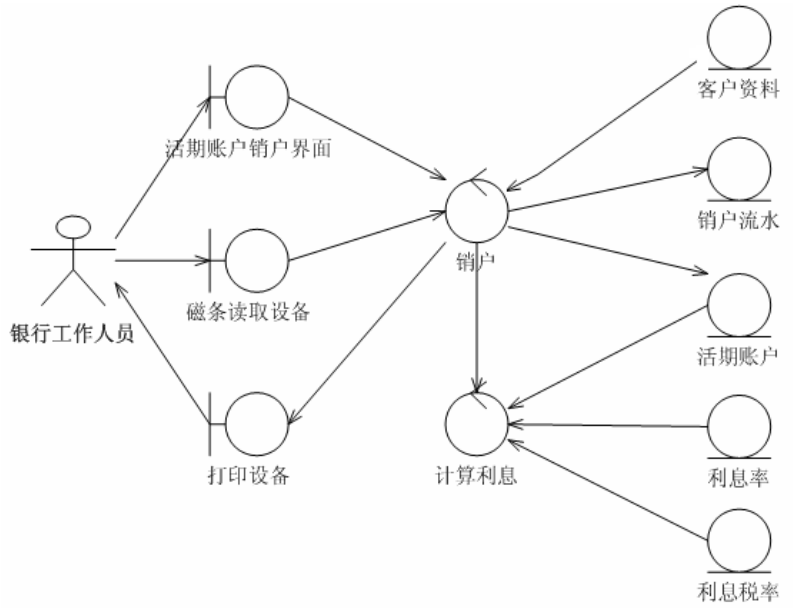


图 9-14 “销户”的鲁棒图

图中的“销户”是一个控制对象，和另一个控制对象“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。

由此例，还想请大家体会另一点实践要领，即“初步设计不应关注细节”：

- “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。

- 每个对象，只标识对象名，都未识别其属性和方法。
- 而且，鲁棒图中无需（也不应该）标识控制流的严格顺序。

画鲁棒图的一个关键技巧是“增量建模。”在笔者的培训课上，有大约一半的学员训练前感叹“建模难”。具体到鲁棒图建模，专门训练了“增量建模”技巧之后，大家颇为感慨——建模技巧的核心是思维方式，掌握符合思维规律的建模技巧可以大大提高实际建模能力。

例如，网上书店系统。如果网上书店的搜索功能不易用、速度慢，一定会招致很多抱怨。下面请大家和我一起来为“按作者名搜索图书”功能进行鲁棒图建模。运用增量建模技巧。

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 9-15 所示，所谓搜索功能，就是最终在“结果界面”这个边界对象上显示“Search Result 信息”。“你”认为“获得搜索结果”是最重要的控制对象，它的输入是“作者名字”这个实体对象，它的输出是“Search Result 信息”这个实体对象。

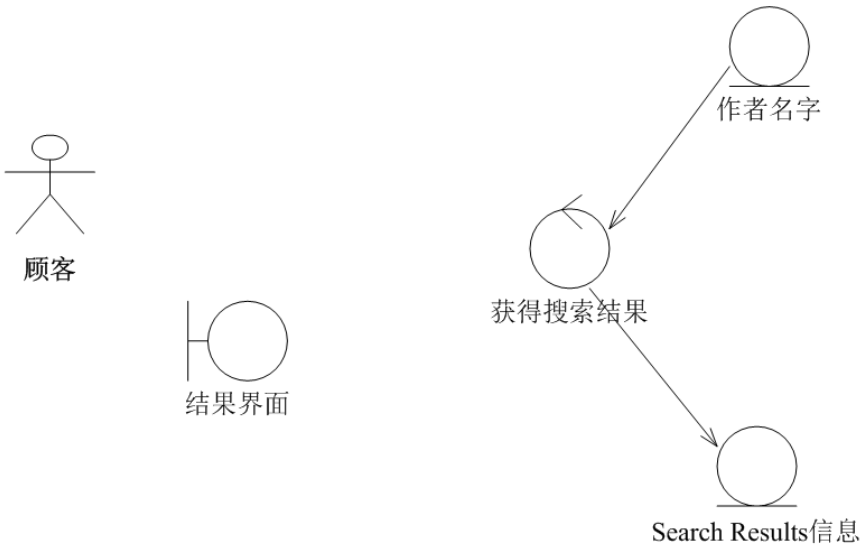


图 9-15　增量建模：先识别最“明显”的职责

接下来，开始考虑职责间的关系，并发现新职责。嗯，“获得搜索结果”这个控制对象，要

想生成“Search Result 信息”这个实体对象，除了“作者名字”这个已发现的实体对象，还要有另外的输入才行，“你”想到了两种设计方式（如图 9-16 所示）：

- 引入“图书信息”实体对象。
- 或者，将“书目信息”和“图书详细信息”分开，前者的结构要专为快速检索而设计以提高性能。……本章稍后利用“目标-场景-决策表”将更清晰地讨论高性能设计。

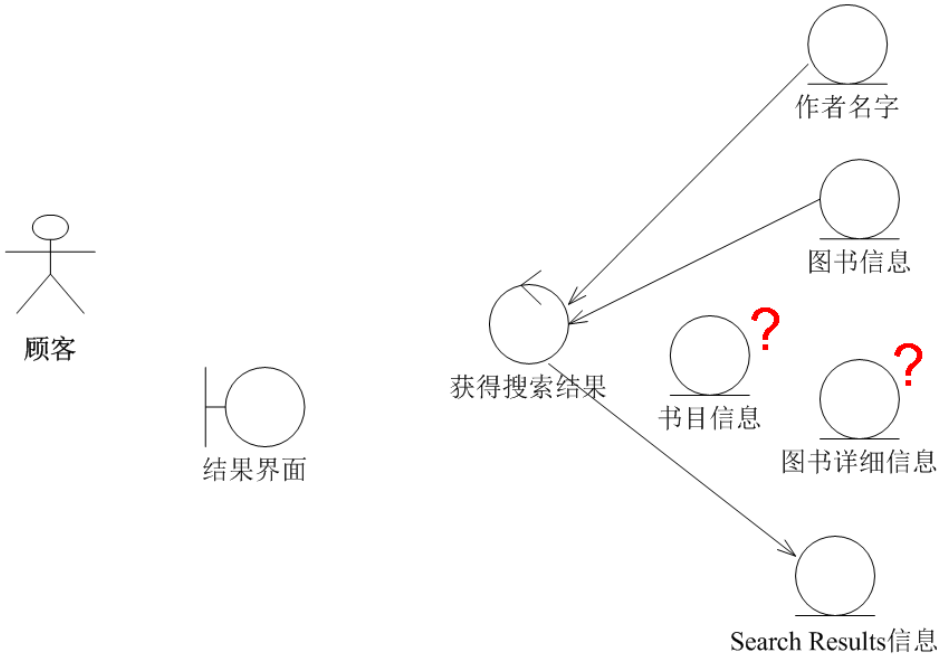


图 9-16　增量建模：开始考虑职责间关系，并发现新职责

继续同样的思维方式。图 9-17 的鲁棒图中间成果，又引入了“显示搜索结果”控制对象，顾客终于可以从边界对象“结果界面”上看到结果了。

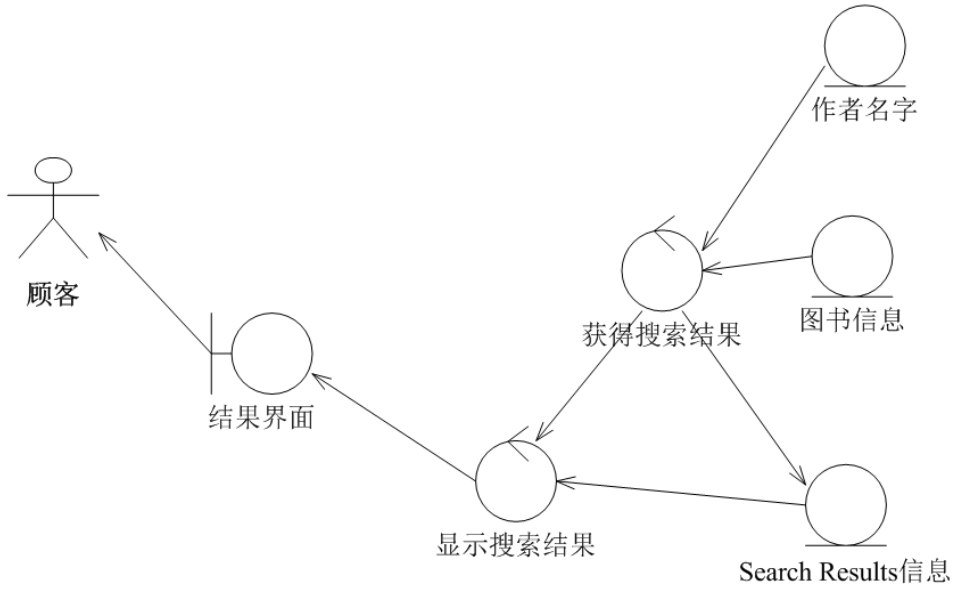


图 9-17　增量建模：继续考虑职责间关系，并发现新职责

增量建模就是用“不断完善”的方式，把各种必要的职责添加进鲁棒图的过程。“不断完善”是靠设计者问自己问题驱动的，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，关键是你的头脑要中。“你”又问自己：

- 根据作者名字搜到了结果，但作者名字从何而来？
- 用户的输入要不要做合法性检查？
- 三个空格算不算作者名字的合法输入？

……最终的鲁棒图如图 9-18 所示。边界对象“Search 界面”被加进来，控制对象“获得搜索条件”和“检查输入合法性”被加进来。

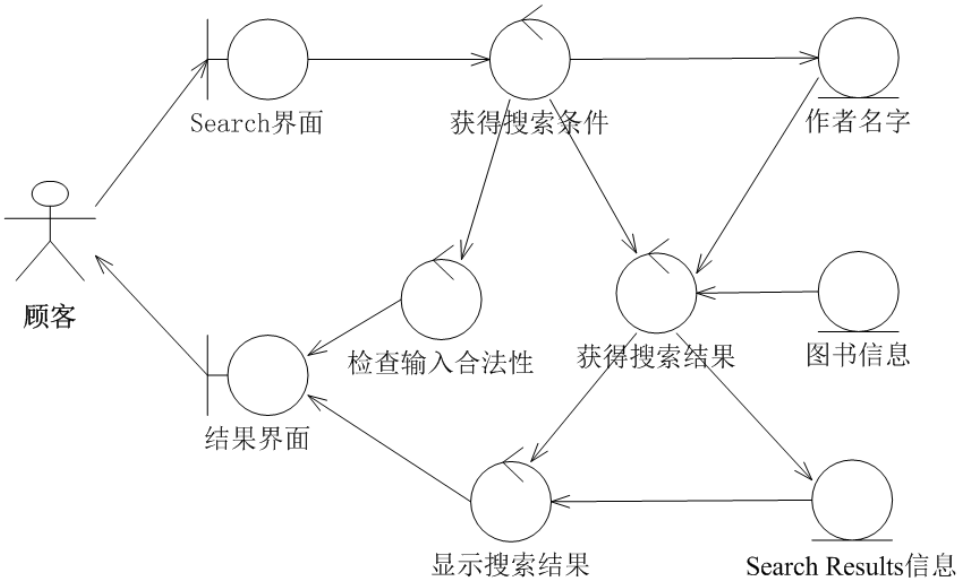


图 9-18　增量建模：直到模型比较完善

9.4　右手质量——概念架构设计（下）

上一节，讲了如何从功能需求向设计过渡，这一节讲如何从质量需求向设计过渡。

9.4.1　什么样的鸿沟，架什么样的桥

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。例如为什么不少软件推出不久就要重新设计（美其名曰“架构重构”）呢？往往是由于系统架构“太拙劣”的原因——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。

然而，从质量需求到软件设计，有个不易跨越的鸿沟：软件的质量属性需求很“飘”，常常令架构师难以把握。例如，根据诸如“本系统应该具有较高的高性能”等寥寥几个字来直接做设计，“思维跨度”就太大了设计很难有针对性。

越过从质量需求到设计的鸿沟，需要搭桥。这“桥”就是下面要讲的场景技术，其关键是使笼统的非功能目标明确化：

9.4.2 场景思维

在软件行业，场景技术有着广泛的应用。如图 9-19 所示，场景是一种将笼统需求明确化的需求刻画技术。

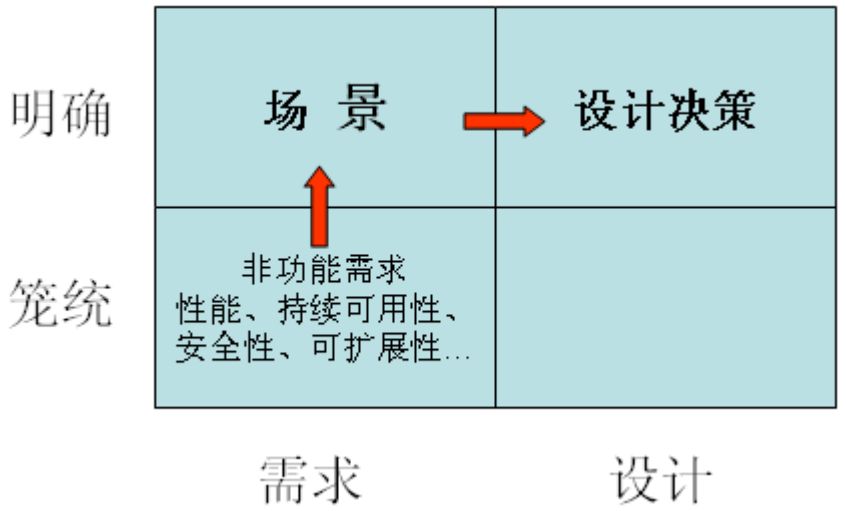


图 9-19　以场景为“跳板”的非功能目标设计思维

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。默认的受影响对象为“本系统”。
- 问题或价值。受影响对象因此而出现什么问题、或需要体现什么价值。
- 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）

9.4.3 场景思维的工具

“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”，软件界泰斗 Edsger Dijkstra 这么说。由此可见，工具的价值。

如果需要，可以借助“场景卡”这种工具，来收集有用的场景——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，这时候架构师可以借助于场景卡来激活团队的力量，让大家提交场景。例如，图 9-20 所示就是一个填写了内容的场景卡。

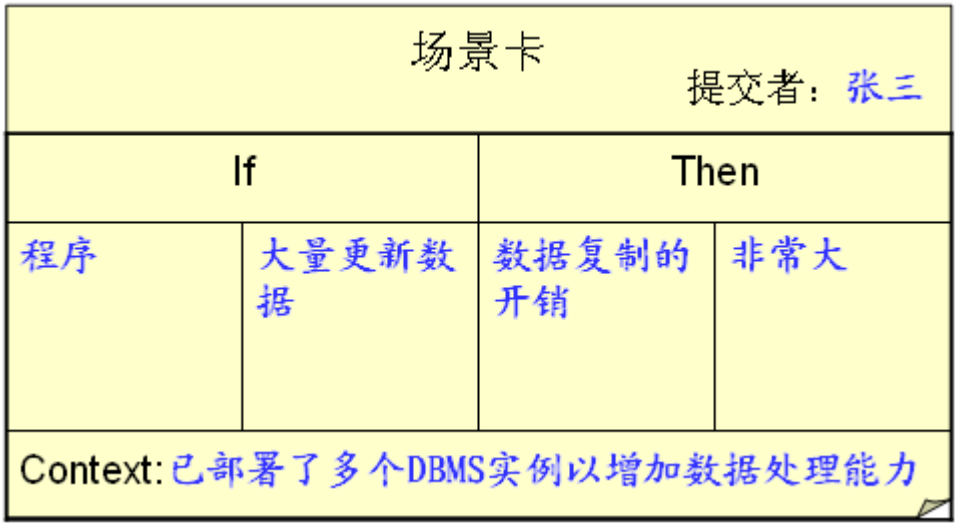


图 9-20　场景卡一例

目标-场景-决策表（如图 9-21 所示）是另一种极其有用的思维工具，熟练掌握大有必要。借助这种思维工具，我们的思考过程形象化了、可视化了。如果说场景卡是“关键点”（用于识别场景），那么目标-场景-决策表就是“纵贯线”（用于打通思维）。

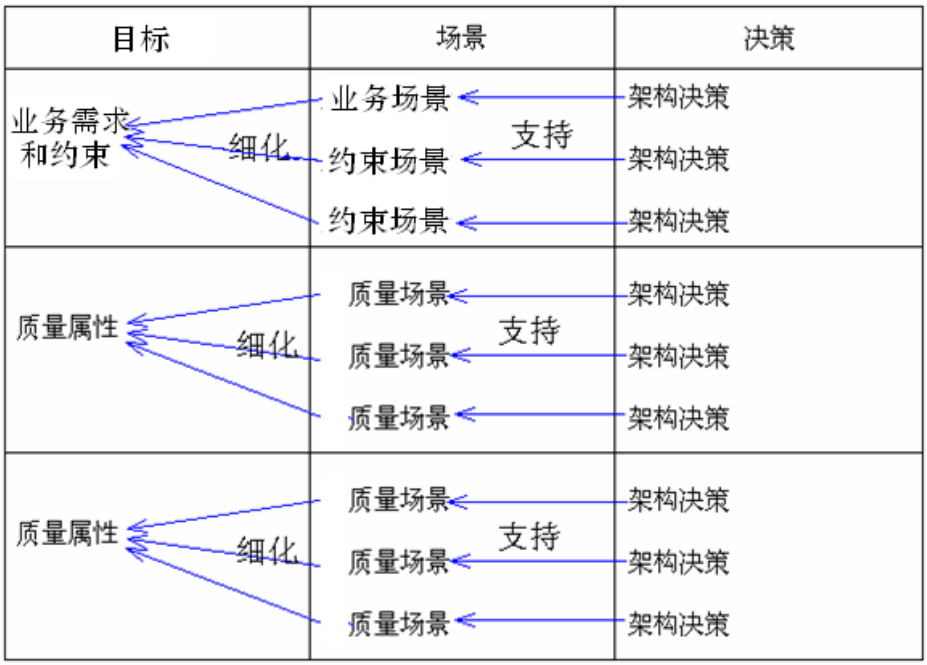


图 9-21　目标-场景-决策表

需要提醒，运用目标-场景-决策表针对质量进行设计时，“不支持该场景”恰恰是一种有价值的决策——如果每个场景都支持，理性设计就无从谈起、多度设计就在所难免了。这就要求我们在实践时，必须对场景进行评估，以决定是否支持这个场景。如图 9-22 所示，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、出现几率等。

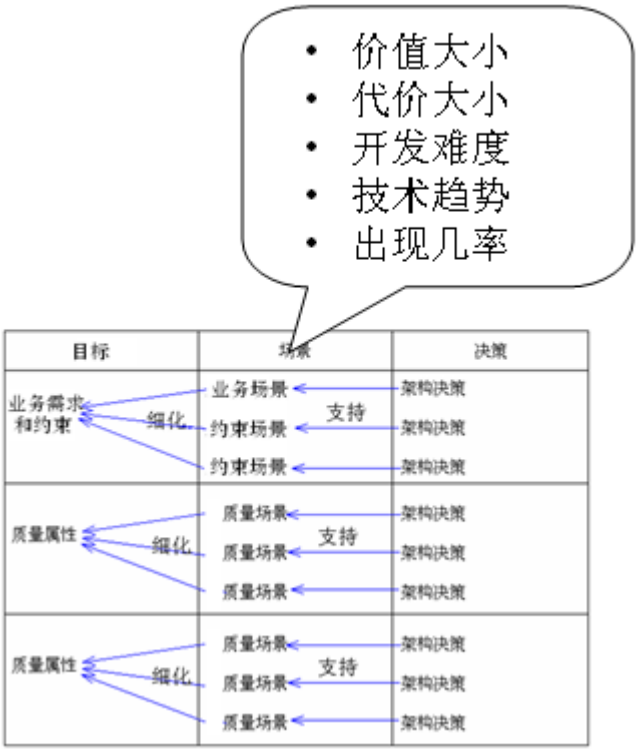


图 9-22　场景评估要考虑的一些因素

9.4.4　目标-场景-决策表应用举例

继续“鲁棒图怎么画”一节的例子——网上书店系统。

如何设计才能使这个系统高性能呢？场景思维是关键。也就是说，我们要明确网上书店系统所处于的哪些真实具体的场景，对高性能这个大的笼统的目标最有意义：

- 如图 9-23 所示，是一个“场景卡”的例子。不断如此明确对性能有意义的“情况”，设计高性能架构也就有“具体的努力方向”了。
- 如表 9-1 所示，我们运用目标-场景-决策表，针对 6 个具体性能场景，设计具体架构决策。

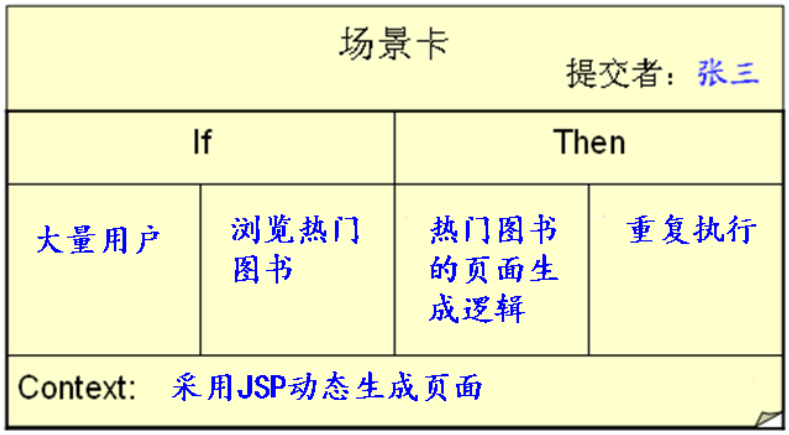


图 9-23　设计网上书店系统：场景卡

表 9-1　设计网上书店系统：目标-场景-决策表

目标	场景		决策
高性能	查询相关	【Context】采用 JSP 动态生成页面 【If】大量用户浏览热门图书 【Then】热门图书的页面生成逻辑重复执行	• 热门图书页面 HTML 静态化
		【If】图书信息一股脑一张表 【Then】搜索图书功能触发大量 IO 开销造成性能低下	• 书目信息和图书详细信息分开保存 • 引入 Cache
	下单相关	【Context】多个用户同时购买同一本书 【If】业务逻辑执行“库存为 0 不能下单”规则 【Then】业务逻辑复杂低效	• 照顾高性能和用户体验，放弃高一致性 • 下单功能有“缺货处理方式”提示 → 等待配齐后发货 → 先发送有货图书
	综合	【Context】业务量增大 【If】系统出现性能瓶颈 【Then】实施群集能否方便高效	• 展现层、业务层、数据层设计成可独立部署的 Tier，方便独立对 WebServer、AppServer、DBServer 做群集
		【Context】业务量增大 【If】计算复杂功能和 IO 量大功能未分离 【Then】部署无法针对性优化	• 将业务层各服务分离，方便优化部署（例如专为高计算量的服务配备高性能 CPU 的机器）
		【Context】新书上架功能写数据库要加锁 【If】上架功能执行中 【Then】查询功能响应慢	• 新书上架功能缺省写入临时表，夜间 3:00 由后台触发自动导入正式表 • 新书上架功能支持操作员指定导入正式表的时机（包括立即写入正式表）

由此可见，通过“目标-场景-决策表”既可以帮助我们引入新的设计（例如表中决策“HTML 静态化”），也可以帮助我们改进了老设计（例如“书目信息和图书详细信息分开保存”，这样前述鲁棒图就可升级为图 9-24 了）。

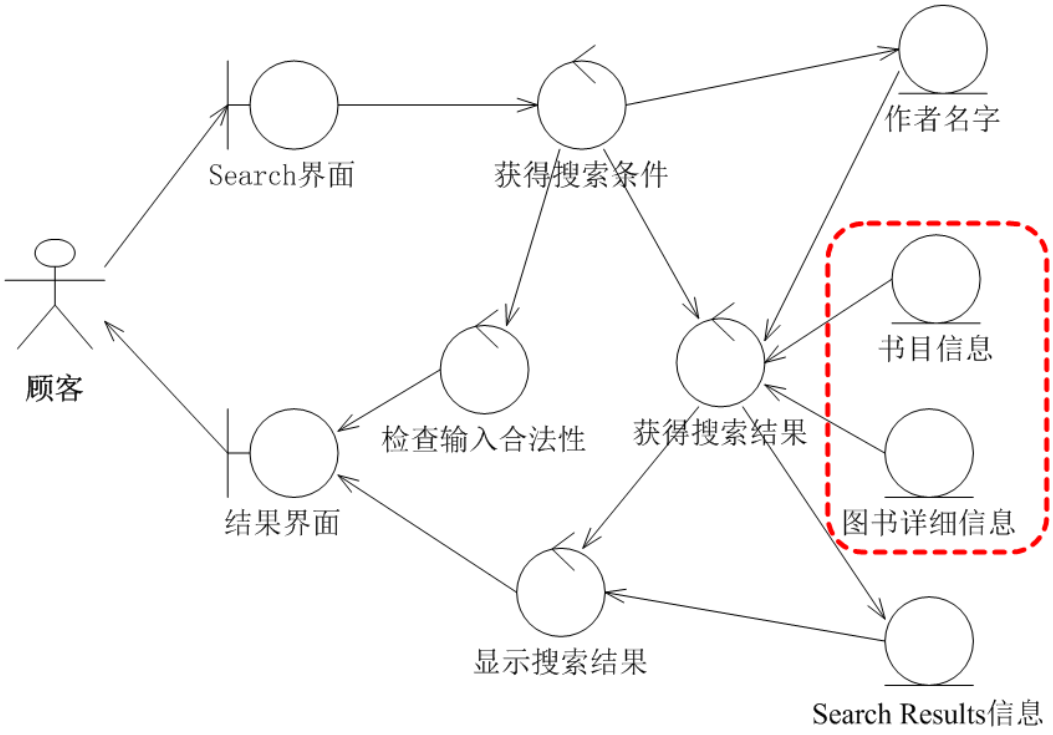


图 9-24　设计网上书店系统：更新设计

9.5　概念架构设计实践要领

9.5.1　要领 1：功能需求与质量需求并重

在“概念架构设计概述”和“左手功能”、“右手质量”等小节，已详细讲解过。

9.5.2 要领 2：概念架构设计的 1 个决定、4 个选择

从设计任务、设计成果上，概念架构设计要明确的是“1 个决定、4 个选择”（如图 9-25 所示）：

- 决定
  - 如何划分顶级子系统
- 选择
  - 架构风格选型
  - 开发技术选型
  - 二次开发技术选型
  - 集成技术选型

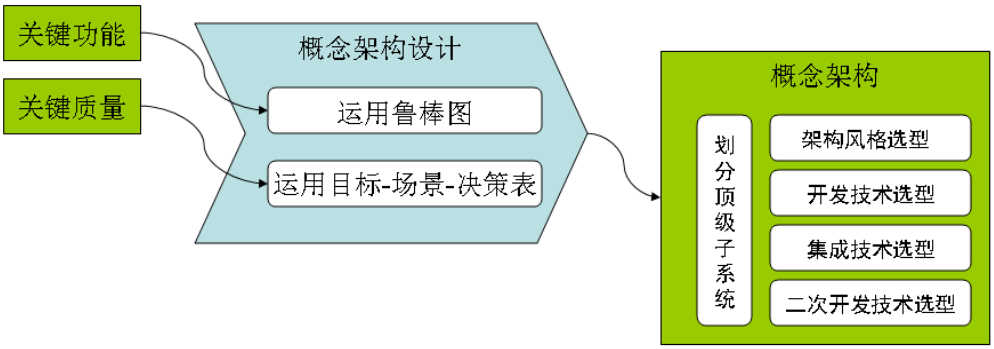


图 9-25　概念架构设计要明确的是“1 个决定、4 个选择”

在实践中，上述 5 项设计任务应该以什么顺序完成呢？笔者推荐（如图 9-26 所示）：

- 首先，选择架构风格、划分顶级子系统。这 2 项设计任务是相互影响、相辅相成的。
- 然后，开发技术选型、集成技术选型、二次开发技术选型。这 3 项设计任务紧密相关、同时进行。另外可能不需要集成支持，也可以决定不支持二次开发。

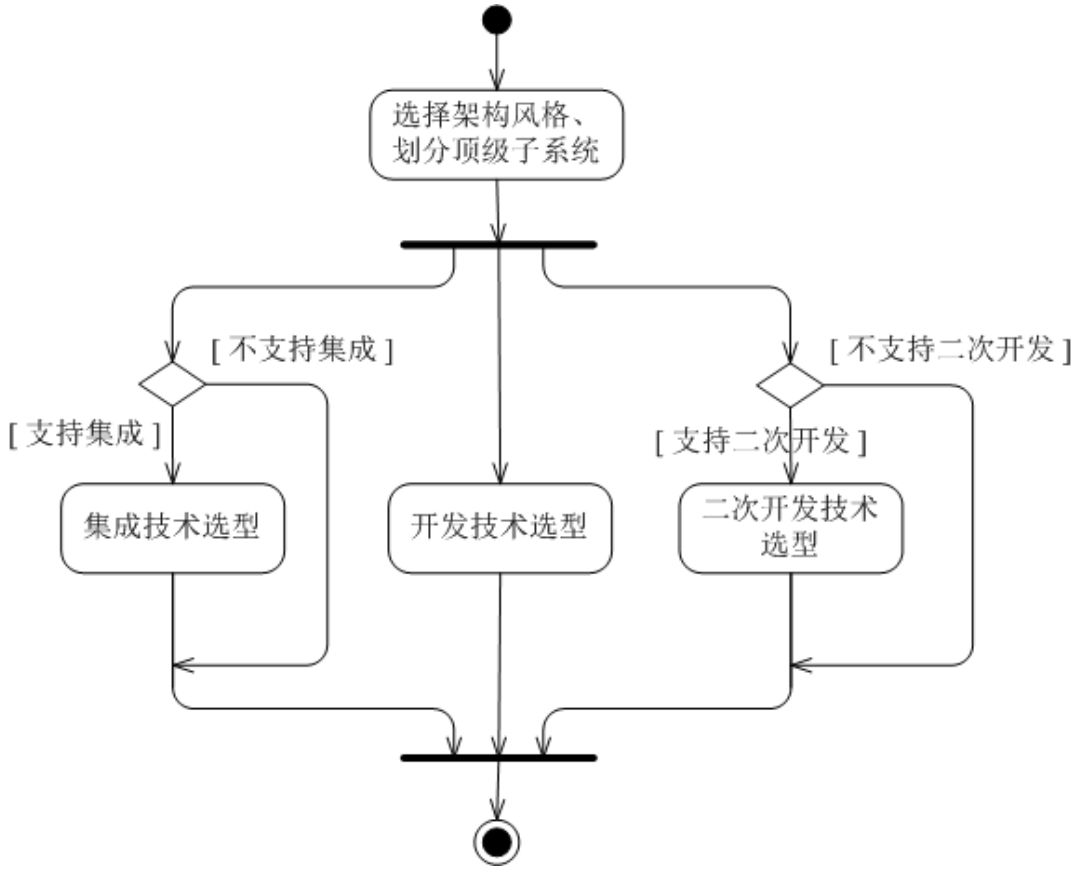


图 9-26　概念架构 5 项设计任务应该以什么顺序完成

9.5.3　要领 3：备选设计

为什么要这样设计？有没有其他设计方式？这都是架构师脑中经常存在的问题。

设计概念架构之时，还处于项目研发的早期阶段，此时不考虑任何可能的备选架构是危险的、武断的，可能造成未来“架构大改”的巨大代价。

笔者推荐一种实用工具，能促进更清晰地评审和对比多个备选概念架构设计方案。如表 9-2 所示，是《概念架构设计备选方案评审表》。

表 9-2 《概念架构设计备选方案评审表》

大项	子项		备选架构 1	备选架构 2	……
设计描述	系统组成	后端			
		前端			
		API			
		插件			
	技术选型	架构风格			
		应用集成			
		UI 集成			
		二次开发技术			
		开发技术			
评审结论	纯技术方案的评价				
	结合企业现状评价				
	选型结果		【 】	【 】	【 】

9.6 实际应用（7）——PM Suite 贯穿案例之概念架构设计

相关技能项，都讲到了。下面实际应用到 PM Suite 贯穿案例，设计 PM Suite 系统的概念架构。

9.6.1 第 1 步：通过初步设计，探索架构风格和高层分割

对于像 PM Suite 这样还算比较复杂的系统，直接确定它为 C/S 架构、或 B/S 架构，那就是拍脑袋。

PM Suite 有哪些功能，这些功能以 C/S 实现合适、还是 B/S 实现合适？

PM Suite 在非功能方面的要求，会如何影响 C/S 架构或 B/S 架构的选择？

为了回答上述极为关键的问题，我们运用鲁棒图对“关键功能”（确定关键需求步骤的输出）进行探索性的初步设计，为真正确定“架构风格和高层分割”积累决策的依据。

项目经理在使用 PM Suite 时，“制定进度计划”是一个关键功能。下面我们开始在鲁棒图的帮助下，通过增量建模探索它的设计，以期达到探索整个 PM Suite 设计方向的目的。

首先，识别你认为最“明显”的职责。如图 9-27 所示。“你”认为制定进度的基本单位是“任务”，因为只有任务进度明确了整个项目的进度计划才得以显现，所以“你”识别出“任务进度设置界面”边界对象、“排定任务时间”控制对象、“任务进度”实体对象。

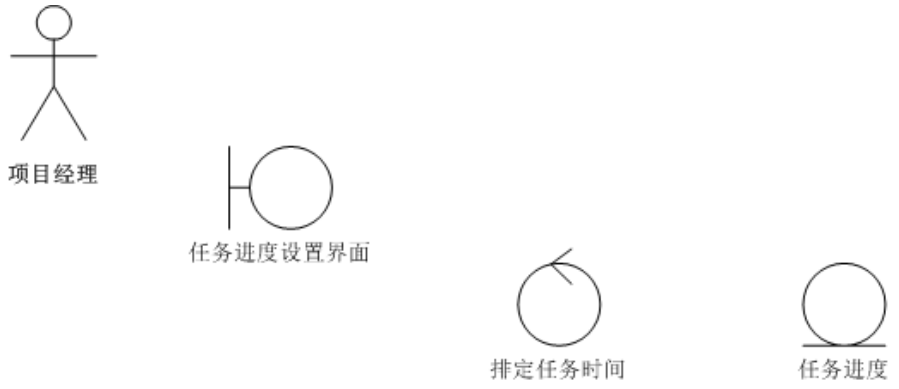


图 9-27 “制定进度计划”的增量建模过程

接下来，开始考虑职责间的关系、并发现新职责，不断迭代：

- 图 9-28。明确已识别的 3 个职责的关系，是“项目经理”通过“任务进度设置界面”触发的“排定任务时间”操作、并引起对“任务进度”实体对象的“写操作”（箭头指向实体对象表示“写”反向表示“读”）。
- 图 9-29。显然，通过“任务进度设置界面”也可以“确定任务依赖”。
- 图 9-30。这时，自然想到，还要支持通过“WBS 展现界面”来“确定任务依赖”。

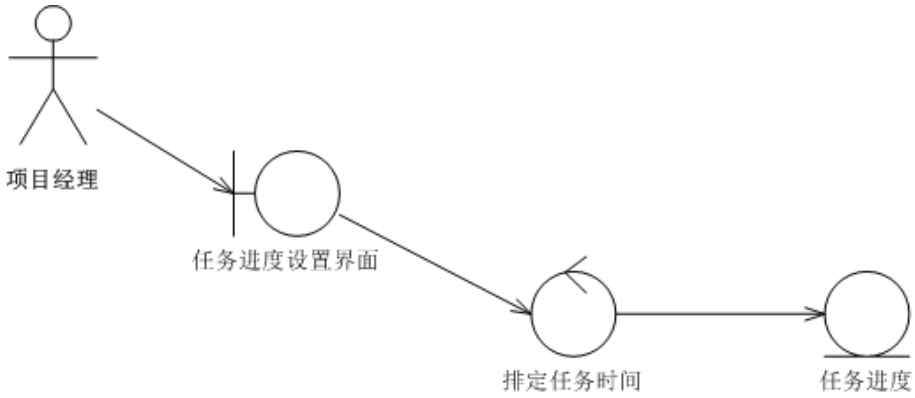


图 9-28　“制定进度计划”的增量建模过程

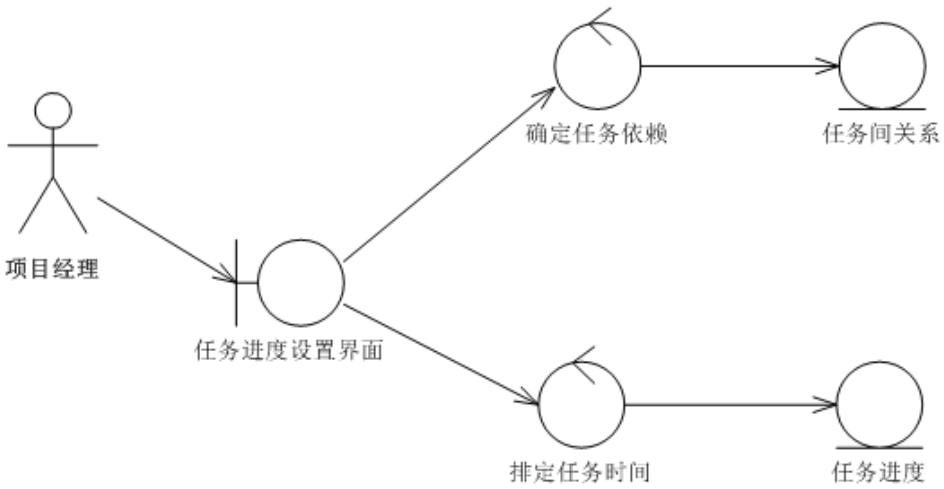


图 9-29　“制定进度计划”的增量建模过程

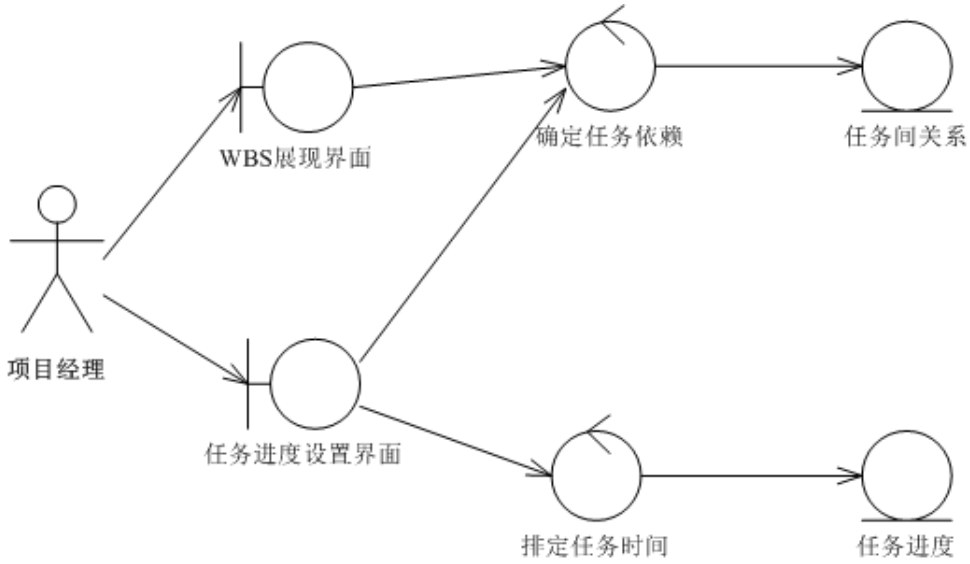


图 9-30　“制定进度计划”的增量建模过程

增量建模技巧的“哲学”是：先把显而易见的设计明确下来，使模糊的设计受到最大程度的启发。

我们再继续迭代，考虑职责间关系、发现新职责、增量地建模：

- 图 9-31。那么，“WBS 展现界面”里的数据从何而来？于是，引入“WBS”实体对象和“更新 WBS 展现”控制对象。
- 图 9-32。“更新 WBS 展现”相关关联比较复杂：
  - 它“读”3 个实体对象：WBS、任务间关系、任务进度。
  - “确定任务依赖”和“指定任务时间”时，都会触发调用“更新 WBS 展现”来刷新界面。

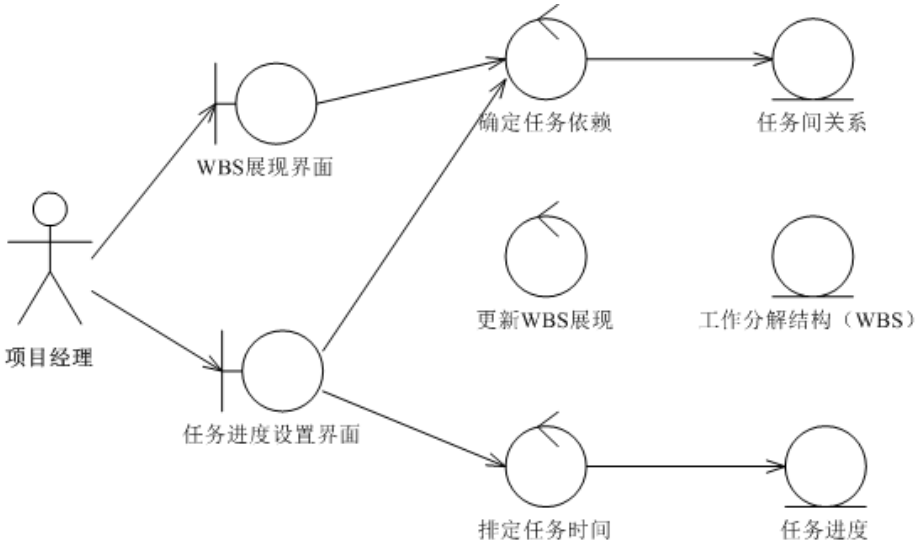


图 9-31　“制定进度计划”的增量建模过程

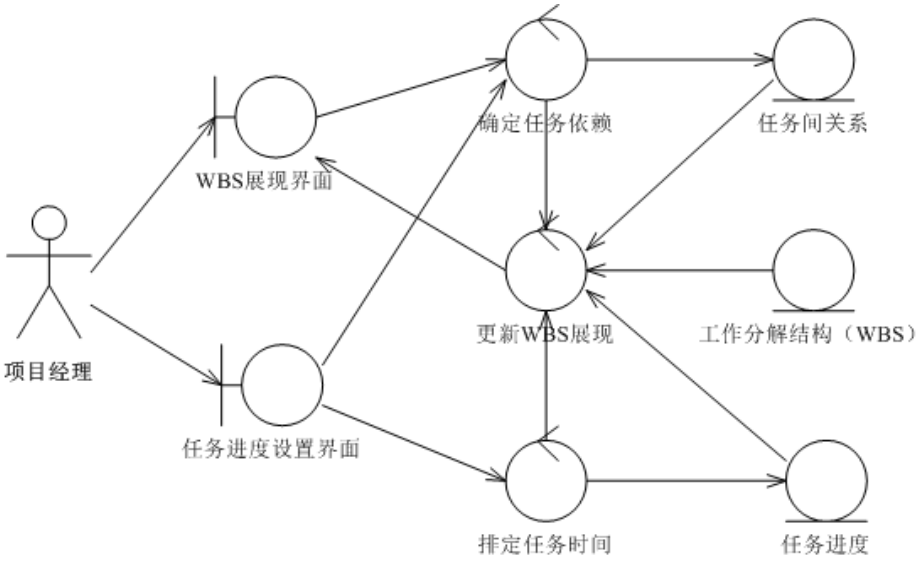


图 9-32　“制定进度计划”的增量建模过程

不断如此分析，一些架构级的“关键决策点”就暴漏无遗了。例如，WBS 的定义是放在 File 中、还是放在 DB 中？如图 9-33 所示。这个问题之所以关键，是因为它和“架构风格选型采用纯 B/S 架构行不行”直接相关。

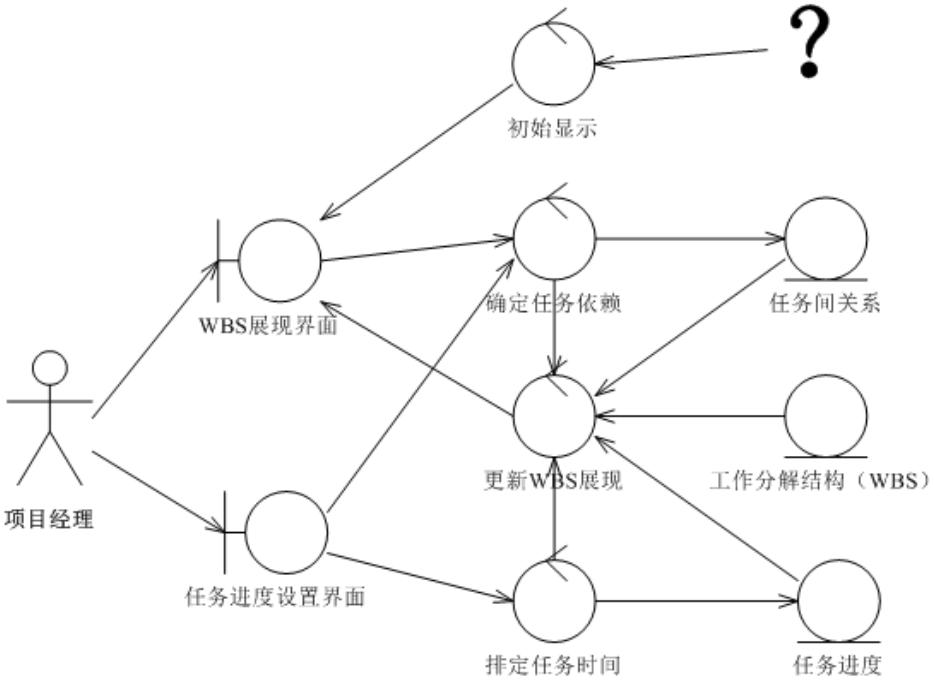


图 9-33　鲁棒图建模能启发架构级的“关键决策点”

值得说明的是，根据系统复杂程度不同、以及架构师的设计经验不同，可以灵活把握对多少“关键功能”进行鲁棒图建模。如图 9-34 所示，是“从 HR 系统导入资源”功能的鲁棒图……

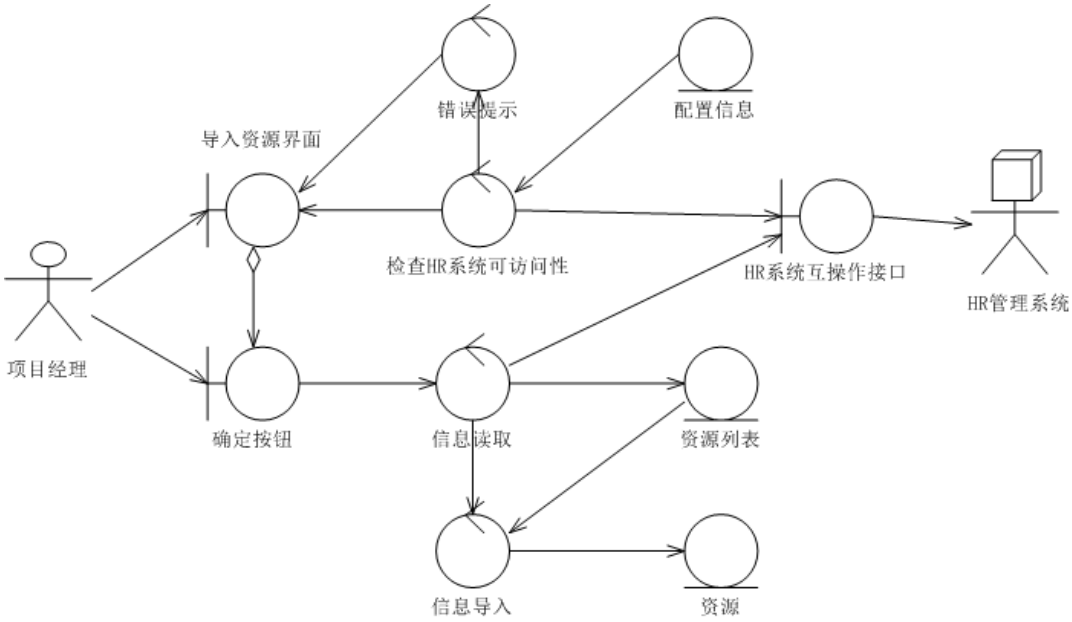


图 9-34　“从 HR 系统导入资源”的鲁棒图

9.6.2　第 2 步：选择架构风格，划分顶级子系统

刚才，我们对一组“关键功能”进行了探索性的初步设计。这，已经为真正确定“架构风格和高层分割”积累决策的依据：

- 这个“制定进度计划”功能，UI 交互比较密集，C/S 架构有优势。
- 一个用久了 MS Project 客户端的项目经理，“制定进度计划”时当然想用类似的方式。于是，架构师考虑是不是能开发类似 MS Project 的客户端呢？如果本公司的客户端并不提供 MS Project 没有的特色功能，能不能直接采用 MS Project 呢？……这些考虑背后，潜在的架构风格决策还是 C/S 架构。
- 但是，很多一般项目成员和高层管理者用到的大量功能，基本上就是“信息展示”，B/S 架构最适合。
- …… 因此，我们决定综合 C/S + B/S 的好处（如图 9-35 所示），结合使用。

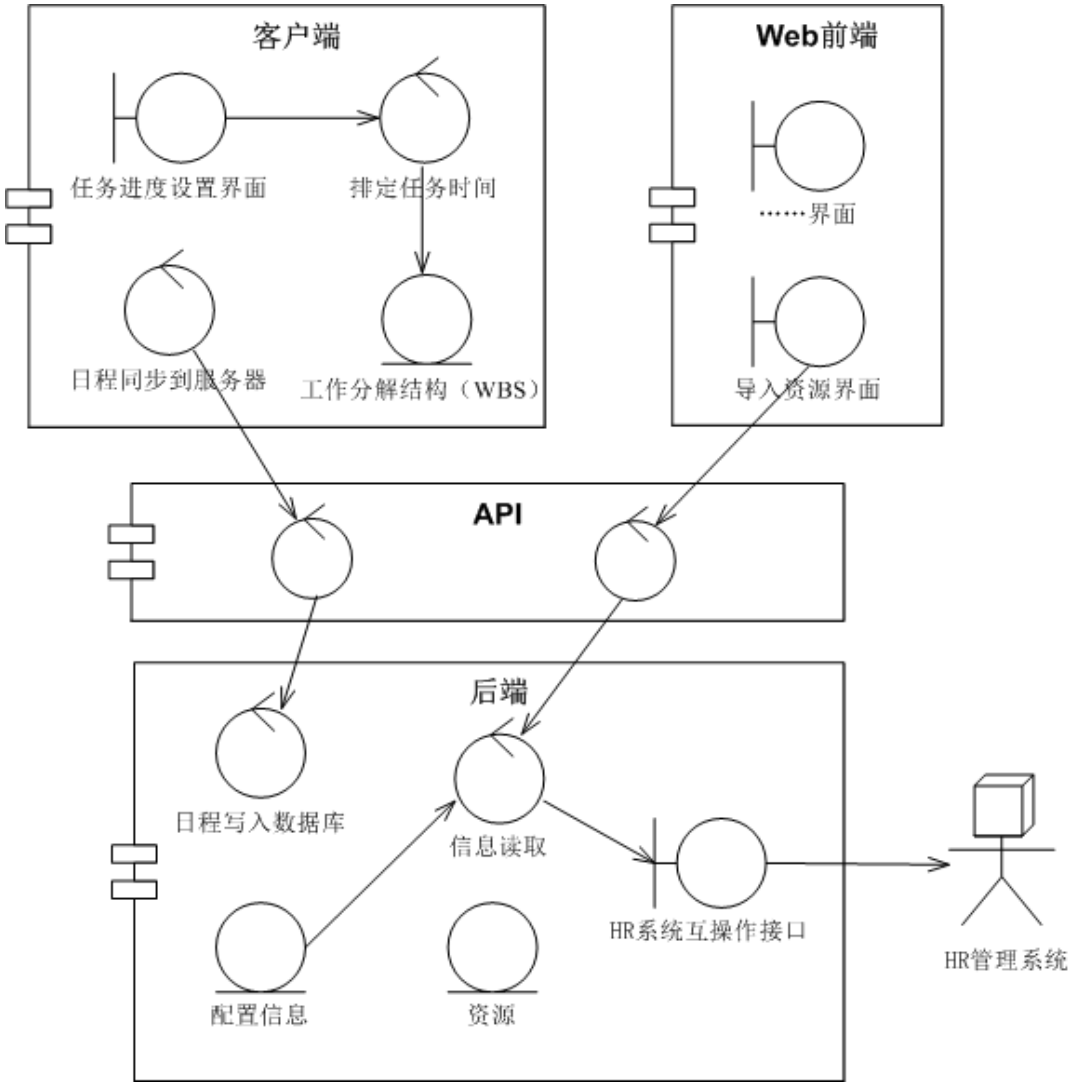


图 9-35　选择架构风格，划分顶级子系统

其中的思维过程，用“目标-场景-决策表”可以更清晰地刻画。如表 9-3 所示。

表 9-3　基于“目标-场景-决策表”思维进行架构风格选型

目标	场景	决策

架构风格选型	【Context】项目经理用久了 MS Project 客户端 【If】没有客户端，完全采用 Web 网页方式提供功能 【Then】项目经理不习惯，难免抱怨	【暂定】开发类似 MS Project 的客户端
	【Context】开发类似 MS Project 的客户端 【If】如果本公司的客户端并不提供 MS Project 没有的特色功能 【Then】开发工作干了不少，但没有商业效果	【最终】直接采用 MS Project 做客户端，后端要支持
	【Context】很多一般项目成员和高层管理者用到的大量功能，基本上就是“信息展示” 【If】如果采用 C/S 架构 【Then】很多客户端需要部署和维护	【暂定】采用 B/S 架构
	【Context】PM Suite 各种功能特点不一 【If】单独采用 C/S 架构，或单独采用 B/S 架构 【Then】总有明显不合理之处	【最终】C/S + B/S 架构

9.6.3 第 3 步：开发技术、集成技术与二次开发技术的选型

一种备选概念架构设计的选型决策如下：

- 开发技术选型。选择 Java 来开发 PM Suite。
- 是否支持二次开发。支持，一是方便我们自己公司在提供整体解决方案时进行“应用集成”，二是提供给其他厂商供他们的产品调用 PM Suite 的功能。
- 二次开发技术选型。暂时只提供 API for Java，以后根据需要再考虑可能提供的其他 API（例如 API for VB 或支持脚本语言的 API）。

- 是否支持集成。必须支持，PM Suite 的特点就是互操作性要求明显，这在“第 8 章 确定关键需求”中也重点分析了。

- 集成技术选型。分析一下，PM Suite 方案存在少数功能，是需要多系统集成完成的；但另外有数量稍多的“将无关的多个功能统一呈现到某角色的工作台之上”情况。因此决定，Web UI 集成 + 应用集成，这两种技术都将采用：
  - 对“将无关的多个功能统一呈现到某角色的工作台之上”情况，当然首选 Web UI 集成方式（如图 9-36 所示），因为利用 Protal 等技术进行集成开发比较便捷，PM Suite 要“新写的代码量”少。
  - 否则，采用应用集成技术，这时 PM Suite 调用要集成的那些系统的 API，然后还是由 PM Suite 在展现层组织界面显示。代码量多。

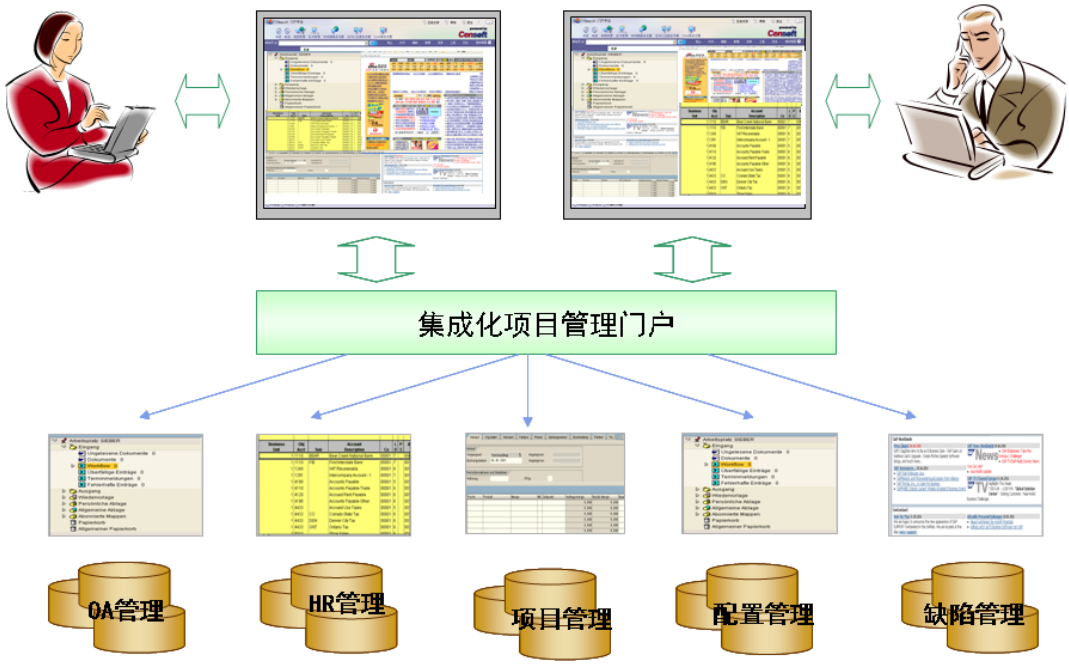


图 9-36　PM Suite 支持集成之 Web UI 方式

9.6.4　第 4 步：评审 3 个备选架构，敲定概念架构方案

决策过程中，总是充满了选择，《三国演义》中曹操兴师也不例外：

- 打刘备，还是打孙权？
- 打孙权的话，我曹操的北方士兵不习水战，怎么办？选择不打了，还是另想办法？
- 还是要打的话，是围而不打，还是主动进攻？
- 主动进攻的话，战船钉在一起会更稳，但选择哪种策略来防备东吴“火烧战船”呢？
- ……“预计不到”和“选择不当”都会造成决策失败！

PM Suite 架构设计的过程，其实也充满了选择。

备选架构 1：一刀切采用 B/S 架构，岂不简单便宜？

备选架构 2：C/S + B/S 混合架构，照顾不同功能的特点。

备选架构 3：PM Suite 要和那么多系统整合，采用基于平台的整合岂不更先进？你看人家 IBM ALM 就基于 Jazz 搞多系统之间的集成。

如果你是架构师，你会怎么办？选择“最牛”的备选架构 3 试试……。如图 9-37 所示，分析了 IBM ALM 的概念架构。针对 PM Suite 的现实需求，架构评审的结论是：照搬 IBM ALM 的概念架构为“过度设计”。

系统组成	后端	多个后端
	前端	多个 Web 前端 多个自研客户端 兼容 MS Project 客户端
	API	多个 API
	插件	Eclipse 插件、 VisualStudio 插件
技术选型	架构风格	C/S + B/S
	应用集成	自研类似 IBM Jazz 的集成平台，包含 Message Broker 等基础设施
	UI 集成	Web UI 集成、Eclipse 及 VisualStudio 客户端集成
	二次开发技术	Java API、VB API、 Perl API
	开发技术	Java、JSP、C++

图 9-37　分析 IBM ALM 的概念架构

笔者推荐基于《概念架构设计备选方案评审表》进行对比、评价。如表 9-4 所示，从“设计描述”的 9 类“子项”的明确，到“评审结论”的最终得出，都一目了然。

表 9-4　运用《概念架构设计备选方案评审表》对比评审 PM Suite 概念架构

大项	子项		【备选架构 1】 纯 B/S 架构	【备选架构 2】 C/S+B/S 混合架构	【备选架构 3】 学 IBM ALM 架构
设计描述	系统组成	后端	前后端一体 Web 应用	1 个后端	多个后端
		前端	同上	1 个 Web 前端 客户端用 Project	多个 Web 前端 多个自研客户端 兼容 MS Project 客户端
		API	无	1 个 API	多个 API
		插件	无	无	Eclipse 插件、 VisualStudio 插件
	技术选型	架构风格	B/S	C/S + B/S	C/S + B/S
		应用集成	无	调第三方系统 API	自研类似 IBM Jazz 的集成平台，包含 Message Broker 等基础设施
		UI 集成	无	Web UI 集成	Web UI 集成、Eclipse 及 VisualStudio 客户端集成
		二次开发技术	无	Java API	Java API、VB API、 Perl API
		开发技术	微软 ASP.NET	Java、JSP	Java、JSP、C++
评审结论	纯技术方案的评价		优点：设计简单	优点：比较强大	优点：很强大、标准化
			缺点：不够强大	缺点：集成没标准	缺点：难度大、成本高
	结合 PM Suite 需求评价		无集成支持	满足需求 集成方式较现实	集成时要第三方系统(甚至已上线)改程序不现实
			结论：设计不足	结论：设计合理	结论：过度设计
	选型结果		【　】	【 选中 】	【　】

对于软企而言，对比备选设计、评审设计优缺点，能避免“管他三七二十一就这么设计啦”

造成的设计偏差，有利于尽早确定合理的架构选型（而不是后期被动地改），还有利于统一团队上上下下对架构方案的认识（详细设计和编程不遵守架构设计的现象在软企中可是屡见不鲜）。

对于个人设计能力的培养，对比备选设计、评审设计优缺点，能拓宽思路洞察设计的“所以然”，是笔者培训中不断证明了的“最佳培训实践”之一。

而且有《概念架构设计备选方案评审表》的支持，实施起来也并不困难。何乐不为。

第 12 章　粗粒度“功能模块”划分

> 为了成功管理大规模软件系统固有的复杂性，有必要提供一种途径把系统分解为子系统……在进行分解并小心定义子系统之间的接口以后，每个子系统就可以独立地设计。
>
> ——Hassan Gomaa，《用 UML 设计并发、分布式、实时应用》

> 目标是把功能相关、耦合度高的对象划分在一个子系统里……倘若参与同一个用例的对象，不是地理上分散的，那它们就是同一个子系统的候选者。
>
> ——Hassan Gomaa，《用 UML 设计并发、分布式、实时应用》

模块划分是架构师的“看家本领”、是架构师岗位的“基本职责”，其重要性无需多说。程序员要向架构师转型，必须重点学习模块划分技能。

掌握模块划分，我们分 4 步走，日常设计工作常用的功能树、分层架构、用例驱动、模块化等技巧都将做专门讨论：

- 粗粒度的“功能模块”划分，应该怎么做？——本章讲
- 如何分层，如何分别封装各种“外部交互”？——第 13 章讲
- 从用例（需求）到模块划分结构（设计）的具体步骤？——14 章讲
- 水平切分(层) ≠ 垂直切分(功能模块) ≠ 通用专用分离。细粒度模块化时，怎样综合利用多种模块划分的手段技巧？——第 15 章讲

12.1　功能树

12.1.1　什么是功能树

作为需求分析手段，功能树（Function Tree）是一种框架性工具，有助于需求分析人员一层一层地选择确定系统必须具有的各项功能（Function）与特性（Feature）。

作为需求分析成果，功能树是一种功能表达结构，它将“功能大类”、“功能组”和“功能项”的隶属与支持关系以“树”的形式呈现出来，非常直观。如图 12-1 所示，是一个功能树的例子（呼叫中心系统）。

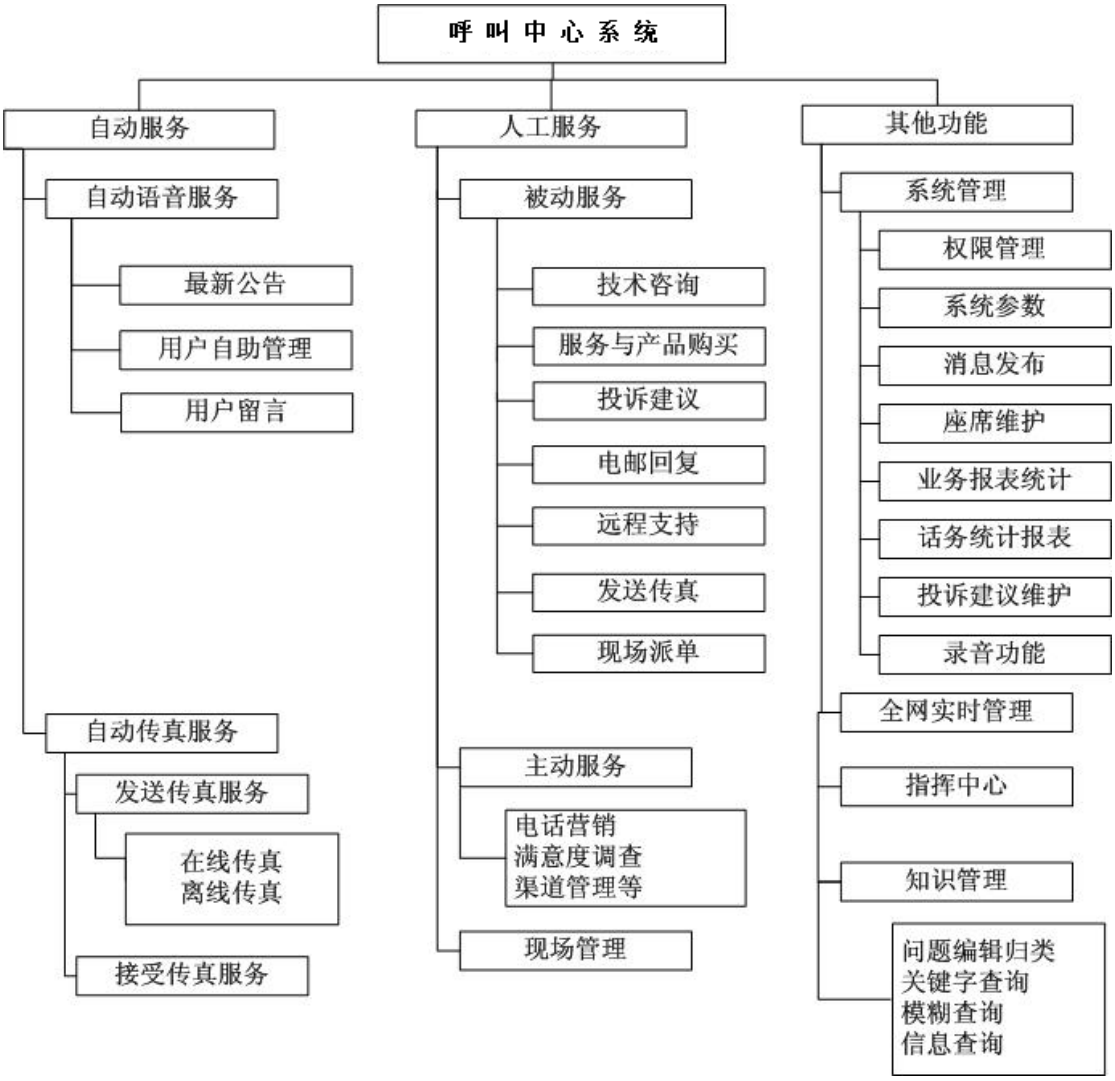


图 12-1　功能树举例（呼叫中心系统）

12.1.2　功能分解≠结构分解

架构设计中的“功能模块”分解，最常见的错误是将“功能分解”与“结构分解”混为一谈。

例如，维基百科上的下述观点是错误的：

When used in computer programming, a function tree visualizes which function calls another.（用于计算机编程领域时，功能树可视化了功能之间的调用关系。）

软件架构设计为什么难？因为它是跨越现实世界（问题领域）到计算机世界（解决方案）之间鸿沟的一座桥（如图 12-2 所示）。需求分析的意图是明确“问题领域”、将要解决的问题以“功能+质量+约束”的形式定义下来。但需求做得再细（例如用例规约中详解描述了各种各样的意外场景、出错场景），也没有打破“系统是黑盒子”这一点。软件架构设计就是要完成从面向问题到面向解决方案的转换，切分结构、定义协作、选择技术……这显然已经“进入了系统黑盒内部”。

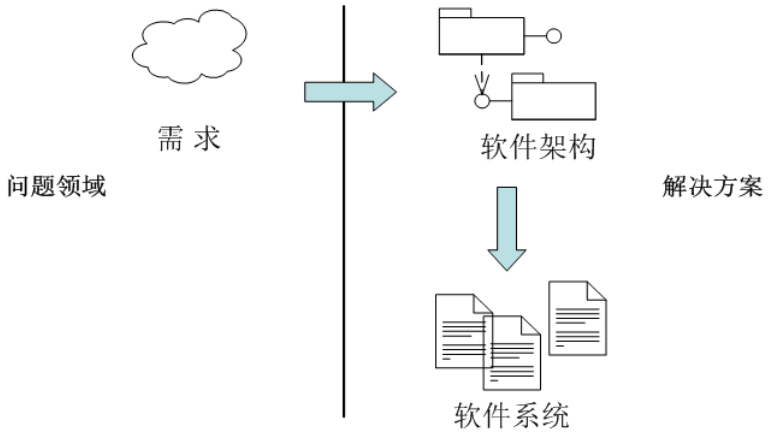


图 12-2　问题领域 vs. 解决方案

所以除非是极其简单的系统，否则必须区分“功能树”和“功能模块结构图”，它们根本不是一回事儿：

- “功能树”是一种功能分解结构，“功能模块结构图”则是对系统进行结构分解；
- “功能树”刻画问题领域，“功能模块结构图”刻画解决方案；
- “功能树”属于需求，“功能模块结构图”属于设计；
- “功能树”是架构师从上游（例如需求分析师那里）得到的，“功能模块结构图”则是架构师要亲自设计出来的。

12.2　借助功能树，划分粗粒度“功能模块”

软件研发是一项环环相扣的系统工程，需求分析到不到位、设计人员会不会充分利用需求文档等成果，对软件研发的成败都至关重要。

现在，你要将系统划分成 N 个粗粒度“功能模块”，那么最直接的帮助就是“功能树”这一需求分析成果。

如何实践呢？

12.2.1 核心原理：从“功能组”到“功能模块”

从“功能组”到“功能模块”，是粗粒度功能模块划分的常见手段。

举例说明。现在，你要设计一个酒店管理系统，它为酒店前台人员、酒店经理、系统管理员等角色提供各种功能，如图 12-3 所示。

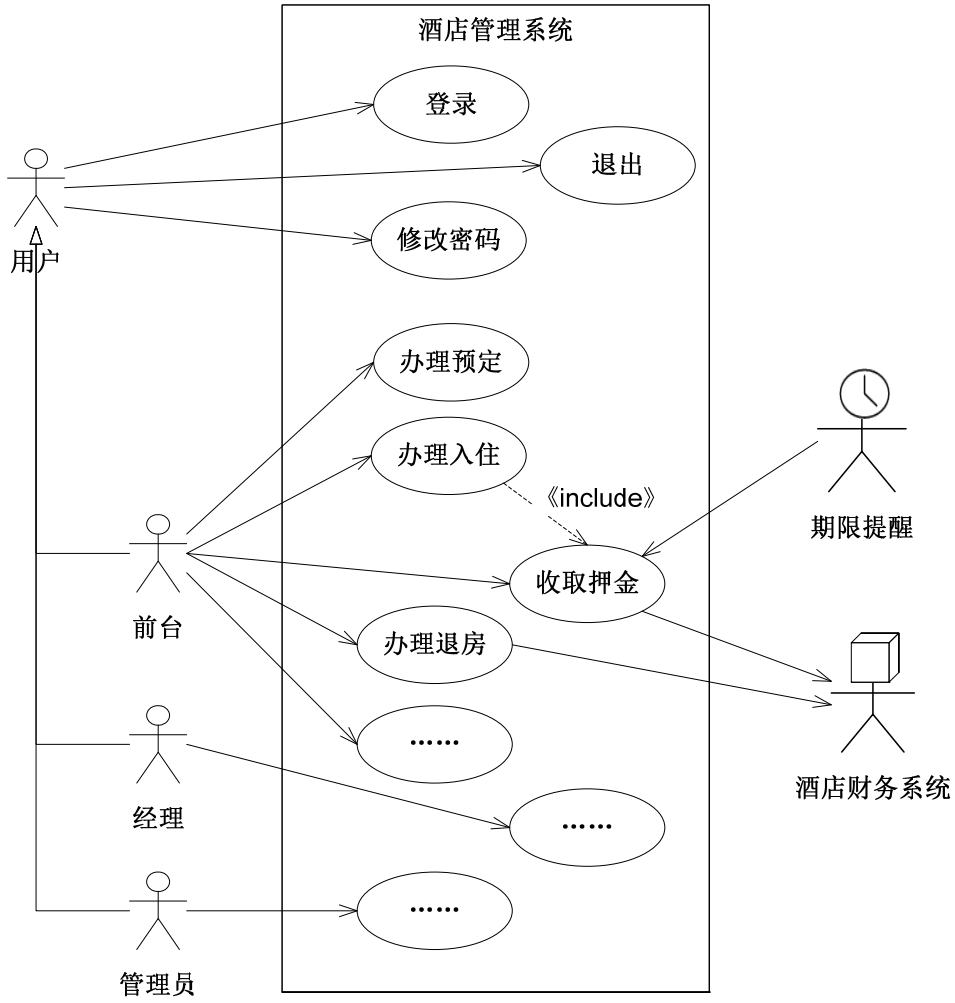


图 12-3　酒店管理系统：用例图

你作为架构师，为该系统设计了如图 12-4 所示的功能模块划分结构。整个系统包含资产管理、宾客服务、人员考核、系统维护等 4 个功能模块（外加 1 个名为角色与权限管理的通用模块）。

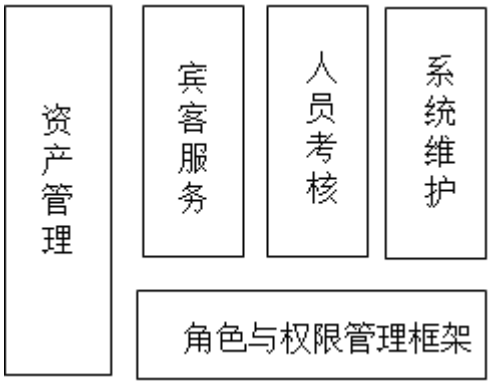


图 12-4　酒店管理系统：功能模块划分结构

但为什么这么设计，设计原理是什么呢？

来分析。每一个具体功能都不是仅由一个类（或一个 C 语言函数）实现的，而是由多个相互协作的不同的软件元素一起完成的。如图 12-5 所示。由图中设计我们可以看出，业务上紧密相关的一组功能的实现涉及到的元素也紧密相关、甚至是同样的元素。例如，“办理预定”、“办理入住”和“办理退房”这三个功能的实现，都涉及 Room 和 Reservation 等类，另外 PayManager 和 PrepayManager 职责相近实现相似（都调用 FinanceProxy），还有 CheckinManager 和 CheckoutManager 也以类似方式处理房间（Room）状态来实现。

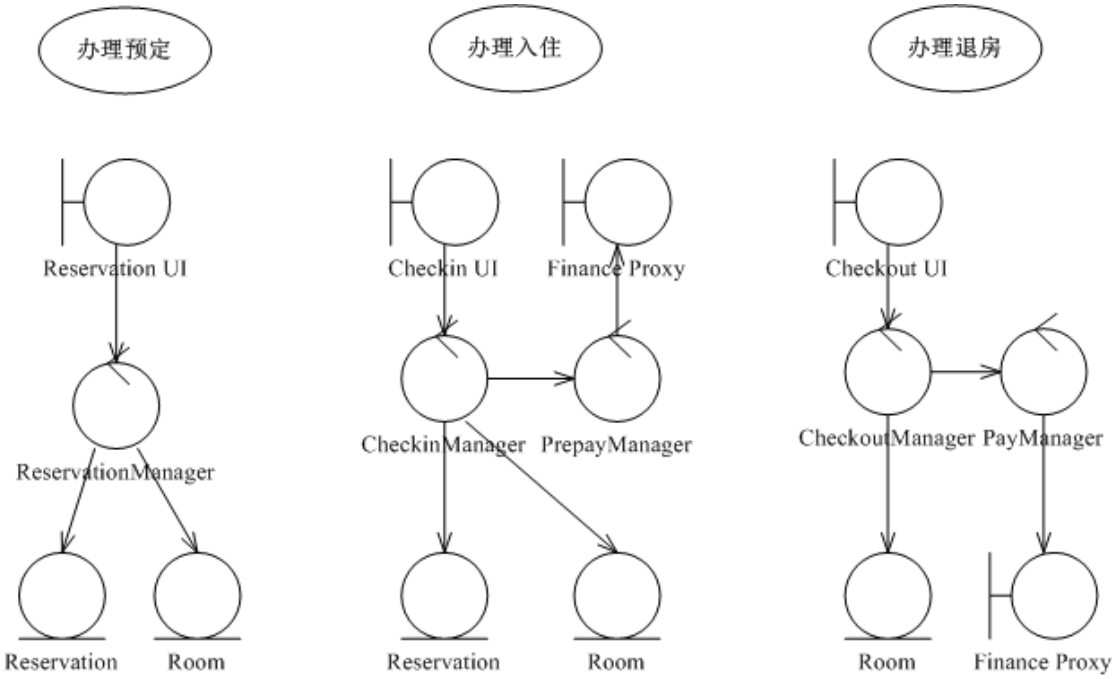


图 12-5　酒店管理系统：3 个功能的实现

于是，将“功能组”对应到大粒度“功能模块”，可以实现模块内的高内聚、模块间的松耦合。例如图 12-6 所示，“办理预定”、“办理入住”和“办理退房”这些相近的功能，可以共享 ServiceManager、PayManager、Reservation 等程序实现——自然，还便于将这组功能分配给一个程序小组负责开发。

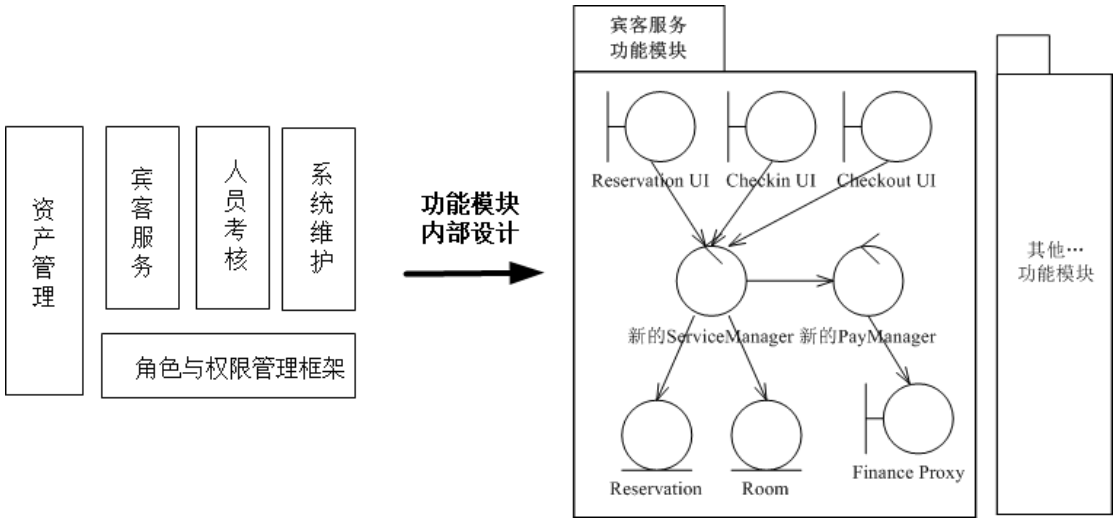


图 12-6 酒店管理系统：粗粒度功能模块结构图

12.2.2 第 1 步：获得功能树

“什么”文档定义功能树？

“谁”能给你或和你讨论功能树？

还有“哪里”有功能树的身影？

要获得需求，总体而言有三种途径：1）拿到文档；2）进行沟通；3）分析产品。

具体来讲：

- 《软件需求规格说明书（SRS）》
  - 《SRS》应该系统化地描述各种功能，是功能树最可能出现的地方。
  - 《SRS》里如果没有“显式的”把功能树画出了，你留意一下它的目录——目录的“章-节-小节”结构经常直观体现了功能树。
  - 如果你希望尽早开始架构设计（《SRS》提交之前），你可以关注更上游的文档、或者借助沟通、或者分析竞争对手的产品。
- 更上游文档
  - 《愿景文档》分析机会、陈述价值、刻画功能体系……，功能树正堪其用。
  - 《方案建议书》梳理需求背景、明确设计原则、刻画系统方案、对比方案优势……，说明系统功能时可能使用功能树。
- 沟通+自画
  - 和业务人员、需求分析人员沟通，获得信息，自己梳理功能树。
- 分析产品
  - 你要设计产品的 4.0 版本，何不研究 3.0 的产品、文档、市场彩页等，获取信息。
  - 当然，也不要忽视竞争对手公司的产品资料。甚至可以运行它，从它的 UI 界面中你可以获得很多帮助，自己梳理功能树。

第一个例子，从需求文档中找功能树。如图 12-7 所示，右边为一份定义良好的《SRS》的部分目录，左边为“功能框图结构”。案例背景是 CRM 系统。

图 12-7　《SRS》是功能树最可能出现的地方

第二个例子，从市场材料中找功能树。如图 12-8 所示，《市场彩页》中，功能树绝对是“常客”。案例背景是酒店综合管理系统。

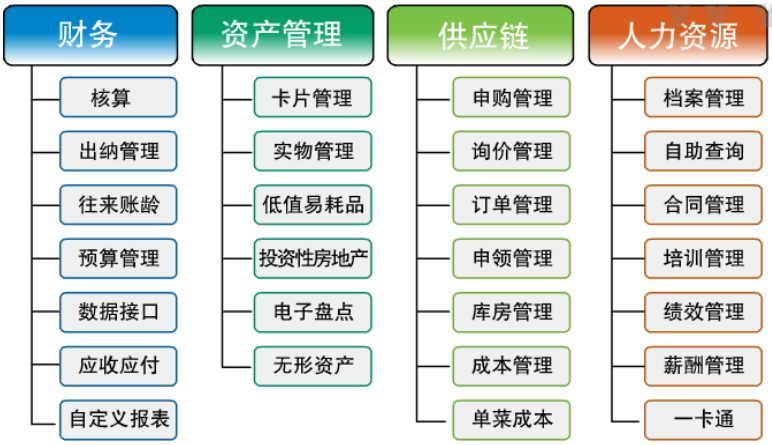


图 12-8　产品的市场彩页中，功能树常见

第三个例子，从方案书中找功能树。如表 12-1 所示，《方案建议书》中全面列举了系统功能。案例背景是汽车生产行业 ERP 系统。

表 12-1　《方案建议书》中的系统功能表

业务类别	部门	关键业务功能
基础管理	各部门	产品目录管理 客户基础管理 仓库基础管理（库区库位） 配送基础管理

		生产线基础数据管理 质量控制标准基础管理 项目基础管理 物料目录信息管理 供应商基础管理
客户管理	市场 财务 物流	合同管理 产品价格管理 订单管理 销售开票管理 销售退换货管理 售后服务管理 销售统计报表
物流管理	物流 市场 生产 财务	订单评审管理 物料需求计划管理 物料库存管理 物料库存预警管理 成品库存管理 配送订单管理 成品发货管理 生产备料管理
生产管理	生产 财务	生产计划管理 生产订单管理 生产统计管理 设备管理
质量管理	质保 物流 生产	材料检验管理 成品检验管理 售后质量反馈 不合格品管理 （评测、退货、返修、拆解、报废处理）
采购管理	物流 财务	采购计划管理 采购订单管理 采购作业管理 供应商考核管理 （供货质量、价格、交货期保障、服务）
财务管理	财务 业务部门	总账、报表管理 应收、应付账管理 资金管理、开票管理 成本核算管理 固定资产管理
人力管理	人力行政 业务部门	组织岗位管理 人员信息管理 系统授权管理 工资管理
项目管理	工程	项目计划与进度管理

	业务部门	项目资源管理
产品数据管理	工程	产品 BOM 配置管理 工程变更管理

第四个例子，分析产品提炼功能树。例如一个设备管理系统（可以是本公司的或竞争对手的），图 12-9 是一般用户的主界面，图 12-10 是系统管理员的权限设置对话框：

- 主界面左边的 Pane，展现给用户的就是一个标准的功能树，树的叶子就是一个个可供用户使用的功能项。
- 权限设置对话框，支持系统管理员为每个功能项设置用户访问权限，也采用了功能树来呈现所有功能项。
- 两个界面给出的功能树是一致的，都包含“设备购置”、“设备台账”、“设备转资”等共 7 个功能组。

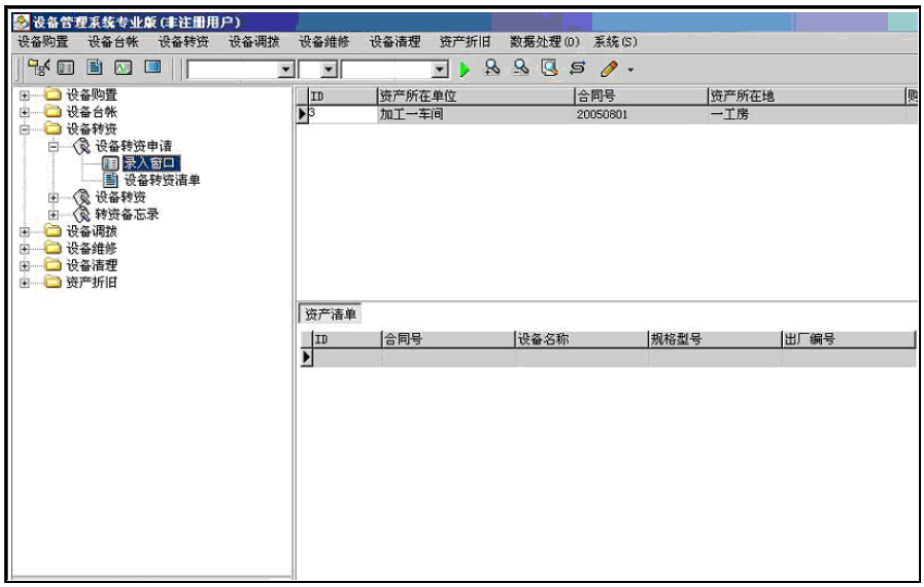


图 12-9 设备管理系统：一般用户的主界面

图 12-10　设备管理系统：系统管理员的权限设置对话框

由上面几例，我们注意到重要一点：功能树除了“树状结构”之外，还可能是“表状结构”、“列表结构”、或者“功能框图结构”等。

12.2.3 第 2 步：评审功能树

希望通过功能树启发自己进行功能模块划分的架构师，首先不能被“假的功能树”骗了。也就是说，架构师得在一定程度上掌握“功能树”这种需求技能：

- 例如设备管理系统，图 12-11 所示的号称是“功能树”的结构，其实是“界面转换图”（若是 Web 系统则称为“页面流程”）。既然根本不是功能树，当然也不能启发“正确的”功能模块划分。
- 对比而言，图 12-12 所示才是功能树。上级节点是高一级“功能组”，下级节点是“细分功能”，上下节点之间是隶属与支持的关系。

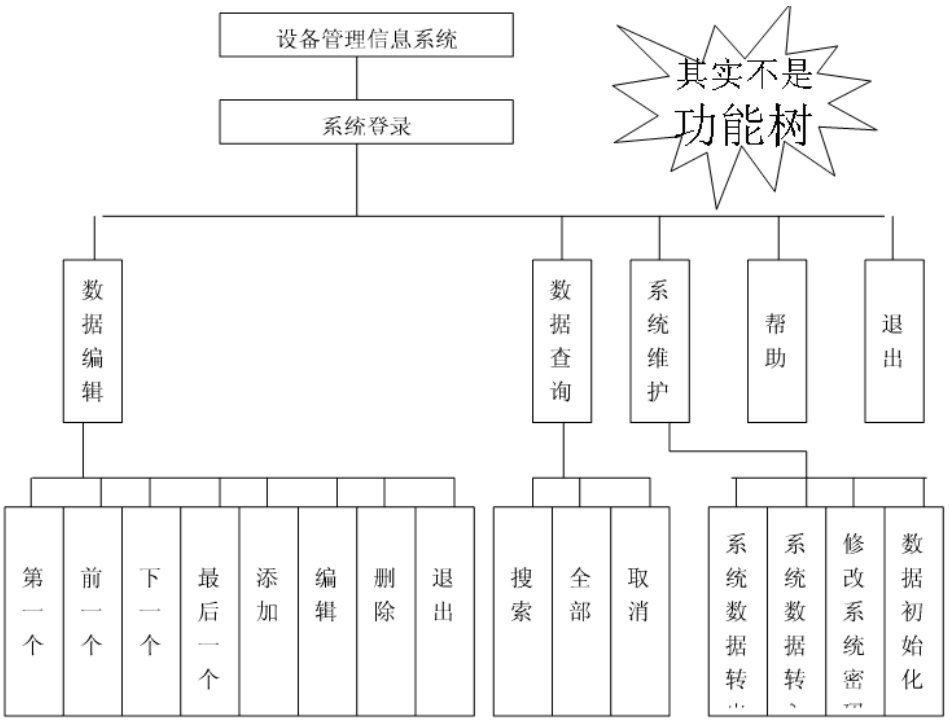


图 12-11　设备管理系统：号称是“功能树”，其实是“界面转换图”

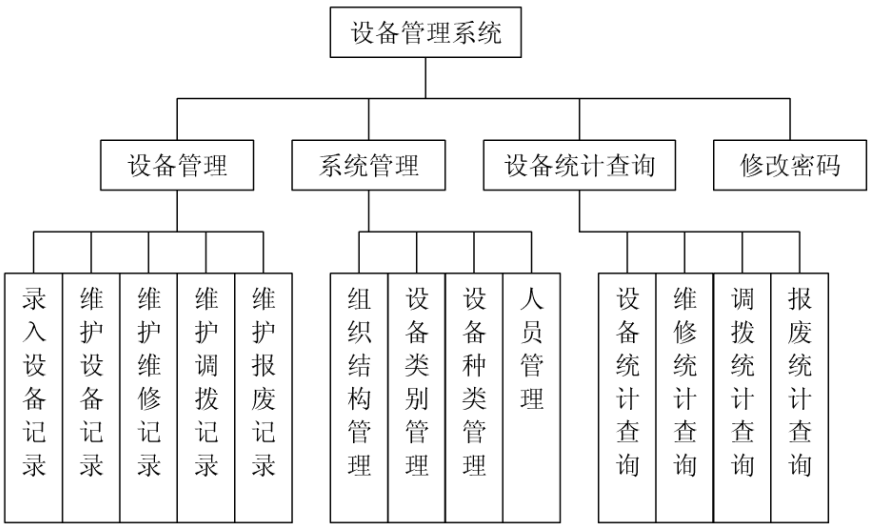


图 12-12　设备管理系统：这个才是“功能树”

那么，什么样的功能树算是“定义良好”的功能树呢？我们应该从如下两个方面评判功能树是否合理（不满足要求的功能树则应进行改进）：

- 一是面向使用、体现使用价值
- 二是覆盖全面、没有范围遗漏

例如，图 12-13 所示，这是一个比较差的功能树，因为它不满足上面“面向使用”和“覆盖全面”的任何一条要求。“金额累加”是功能吗？“故障报警”功能有吗？上货人员的功能何在？钱匣要不要控制？哪个“功能”（需求）能启发架构师发现“钱匣控制”模块（设计）？……如果一个更复杂系统的功能树这么画，岂不是死定了！

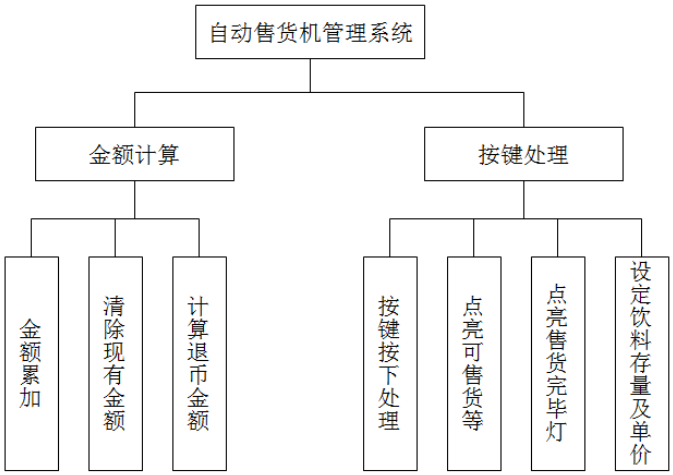


图 12-13　自动售货机管理系统：一个比较差的功能树

12.2.4 第 3 步：粗粒度“功能模块”划分

如前两步所述，架构师知道如何同需求人员打交道、如何评价拿到的需求，是架构岗位本身应有

的技能，是架构师作为“通才”的一个例证。

下面，就是划分“功能模块”技能了。

如图 12-14 所示。一方面，业务上紧密相关的一组功能在实现上也常会涉及相同的函数、类、数据结构等，因此我们应该将“这组功能”映射到一个“功能模块”，这样有利于设计的高聚合、松耦合，还有利于程序小组之间的分工。

另一方面，一些公共服务（例如报错处理、Log 日志、安全验证）会用于同时支持多组功能的实现，它们不单独属于任何“功能模块”，应当进行独立的模块化——将这些公共服务分别放入相应的“通用模块”或“通用机制”中。

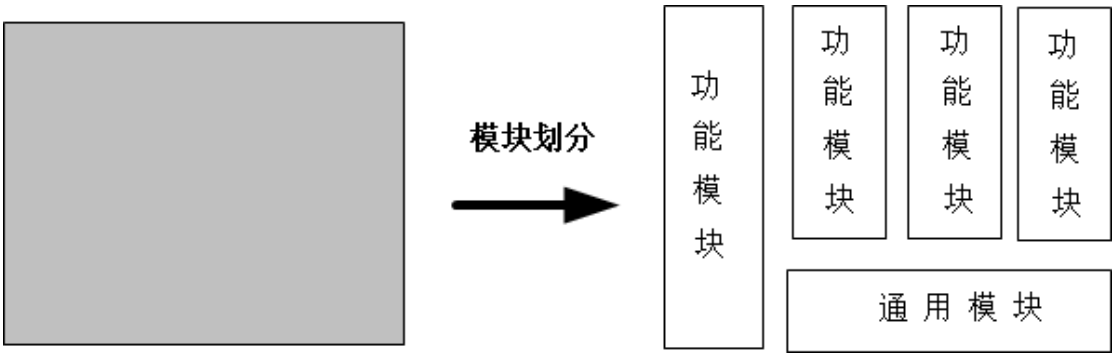


图 12-14　粗粒度功能模块划分，并分离通用模块

12.3　实际应用（9）——对比 MailProxy 案例的 4 种模块划分设计

12.3.1　设计

12.3.2　设计的优点、缺点

第 14 章　用例驱动的模块划分过程

> Use Case 仅提供了设计所需要的所有黑盒行为需求。
>
> ——Alistair Cockburn,《编写有效用例》

> “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》

用例技术，是功能需求实际上的标准。应用用例技术的企业和个人都非常多。既然如此，深刻领会如何从作为需求的用例、过渡到模块划分设计，就大有现实意义了。

这就是本章的主题——用例驱动的模块划分过程。

14.1　描述需求的序列图 vs. 描述设计的序列图

本节解决 2 个问题：

- 首先，说明描述需求的序列图、与描述设计的序列图之不同。它们一个是刻画“内外对话”，另一个是刻画“内部协作”。
- 然后，关注《用例规约》的事件流说明，理清序列图和需求之间的“渊源”。能打通需求到设计的转换环节，离设计的成功就不远了。

14.1.1　描述“内外对话” vs. 描述“内部协作”

如图 14-1 所示，这是一个典型的描述设计的序列图。不难看出，需要 obj1、obj2、obj3 这三个对象进行“协作”才能完成特定功能（用例）：

- obj1 负责接收来自系统外部的 actor 的请求，如图中“消息 1”所示。例如，用户（人）点击了 UI 界面按钮。例如，网关（系统）发送“呼叫信令”到呼叫处理系统。
- 然后，obj1 会调用 obj2，如图中“消息 1.1”所示。
- 之后，obj2 处理完毕返回，如图中“消息 1.2”所示。
- 此时，obj1 又获得了控制权，它处理完毕后会响应外部 actor，如图中“消息 2”所示。例

如，UI 显示操作结果（给人）。例如，呼叫处理系统发送应答（给外部系统）。

- ……

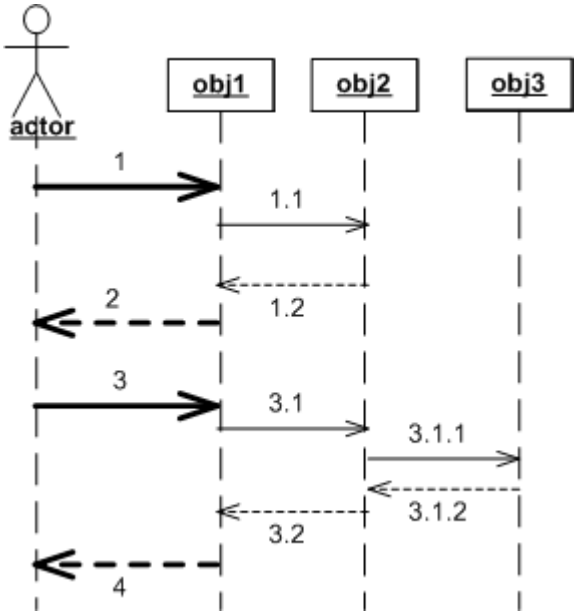


图 14-1　描述设计的序列图

接下来，为了从设计“追根溯源”到需求，我们问自己：描述设计的序列图中，哪些“交互”来自需求级的用例，哪些“交互”出自架构师的设计呢？

如图 14-2 所示，我们在序列图中明确显示出“系统（System）”的位置。如此一来，就一目了然地看出，“消息 1”、“消息 2”和“消息 3”来自需求，因为它们刻画的是外部 actor 和 system 的“对话过程”，经典的方式是在《用例规约》的“事件流”小节描述；而“消息 1.1”、“消息 1.2”、“消息 3.1”和“消息 3.1.1”等才是出自架构师的设计。

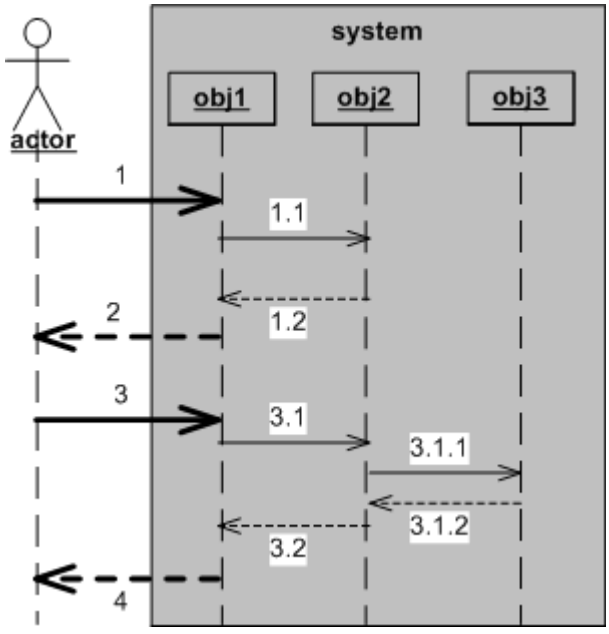


图 14-2　描述设计的序列图中，一半“交互”来自需求

来总结对比。

如图 14-3 所示：左边是描述需求的序列图，右边是描述设计的序列图。我们看到，需求级的、在用例规约中刻画的“Actor 和 System 黑盒的交互序列”，在设计时要被真正支持起来——变成了设计级的、在序列图中刻画的“Actor 和 N 个设计对象的交互序列”。这正是架构师根据用例，进行“用例驱动”设计时要做的核心工作。

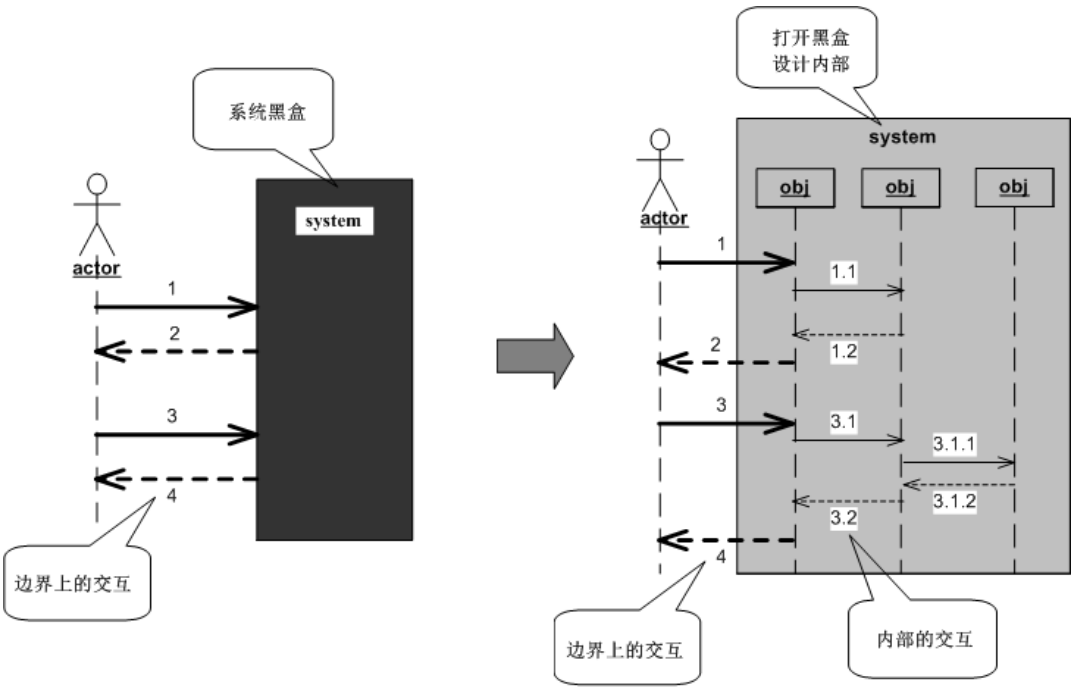


图 14-3 从用例到设计过渡的关键：从“内外对话”到“内部协作”

描述需求的序列图，描述的是“内外对话”。

描述设计的序列图，描述的是“内部协作。

从“内外对话”到“内部协作”，打破黑盒、设计内部，就是从用例到设计过渡的关键。

14.1.2 《用例规约》这样描述“内外对话”

我们经常见到《用例规约》，是采用文本方式，来描述事件流的。如图 14-4 所示，用例规约中的事件流描述，每个句子的“主语”：

- 要么是“各种外部 Actor”。（图中是“银行工作人员”和“客户”）
- 要么是作为黑盒的“待开发系统”。（图中是“系统”）

整体来看，事件流描述了一幅“内外对话”的场景。

1. 用例名称：

   销户

2. 简要说明：

   帮助银行工作人员完成银行客户申请的活期账户销户工作

3. 事件流：

   3.1 基本事件流

   1) 银行工作人员进入“活期账户销户”程序界面；
   2) 银行工作人员用磁条读取设备刷取活期存折磁条信息；
   3) 系统自动显示此活期账户的客户资料信息和账户信息；
   4) 银行工作人员核对销户申请人的证件，并确认销户；
   5) 系统提示客户输入取款密码；
   6) 客户使用密码输入器，输入取款密码；
   7) 系统校验密码无误后，计算利息，扣除利息税（调用结息用例），计算最终销户金额，并打印销户和结息清单；
   8) 系统记录销户流水及其分户账信息。

   3.2 扩展事件流

   ① ……

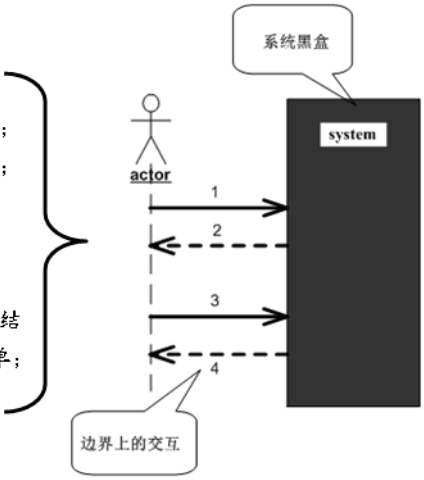


图 14-4　确认：用例规约和序列图的渊源

如果我们乐意，完全可以不用文本方式、而用序列图方式刻画事件流。这时序列图展现的，依然是“待开发系统”和“各种外部 Actor”之间的“内外对话”。

至此，我们已洞察从“用例规约”表达的需求（无论是文本事件流方式、还是序列图事件流方式），向“序列图”表达的设计过渡的关键思维。

14.2　用例驱动的模块划分过程

第 12 章，我们讨论了粗粒度“功能模块”划分，功能相近的一组用例一般被划分到了同一个“功能模块”。……但这，还远远不够，也算不上“用例驱动”的设计思想。

下面，我们看看“用例驱动”的本质思想，讨论从用例、到类、到模块划分的关键思路。

14.2.1　核心原理：从用例、到类、到模块

回顾问题。用例是需求，架构是设计。用例驱动的架构设计，从用例，到模块划分结构，是怎么过渡的呢？（如图 14-5 所示）

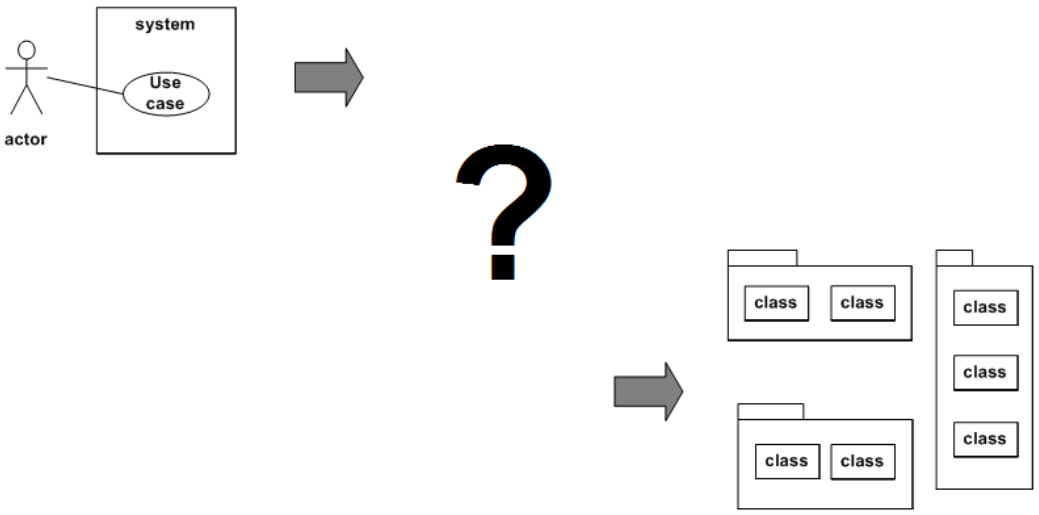


图 14-5　从用例，……到模块划分结构

关键过渡是一组对象的相互协作。所谓协作，是一组对象为了实现某个目的而进行的交互。这里，实现“某个目的”就是实现“用例”定义的功能，而“一组对象”就是用例实现所需要的那些程序元素。嗯！我们最熟悉的序列图，就是围绕一个功能的一组对象的协作。

如图 14-6 所示，从用例，中间经过序列图建模识别出一系列对象，这些对象的合理分组就可促进我们定义出系统的模块结构。

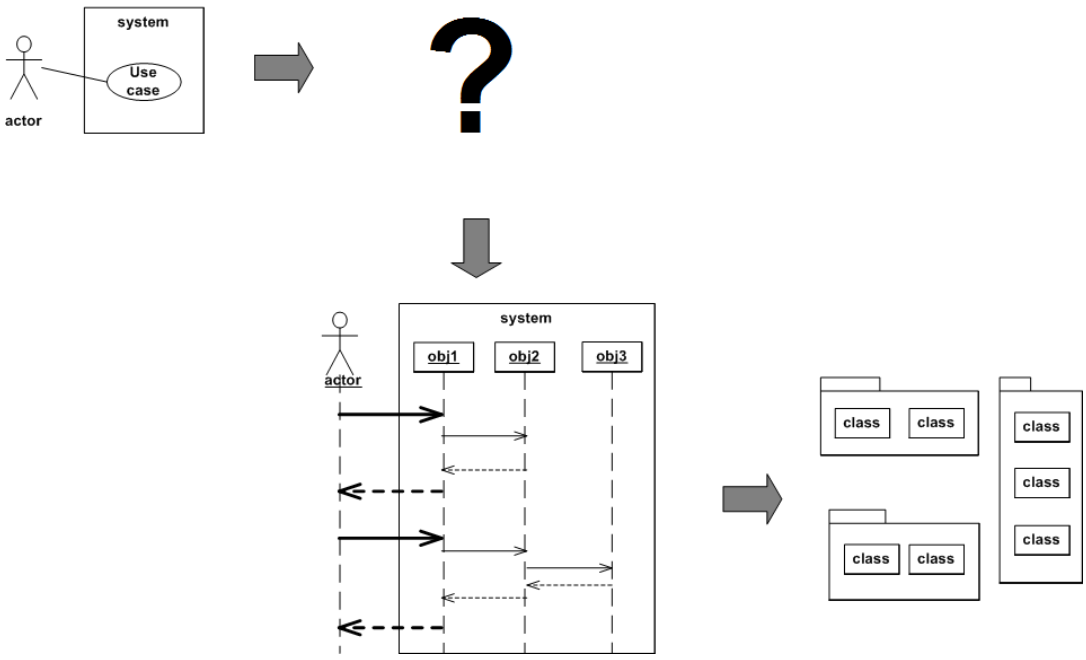


图 14-6　从用例，到一组对象的协作，到模块划分结构

现在，将思路补充完整。如图 14-7 所示，刻画了我们从“需求层”到“设计层”的总体思维路径，简称“两环节、四步骤”：

- 需求分析环节

- 一方面用例图定义系统能提供给外部 Actor 的功能，此步在先；
- 另一方面，用例规约进一步将笼统的“功能”明确定义为“能够为用户带来价值的交互序列”，但仍保持系统作为“黑盒”，此步在后。

■ 架构设计环节

- 一方面，打破“黑盒”，识别一个用例背后有哪些类、设计类之间的交互，此步在先；
- 另一方面，梳理通过多个用例识别出的这些类、将它们划分到不同模块，此步在后。

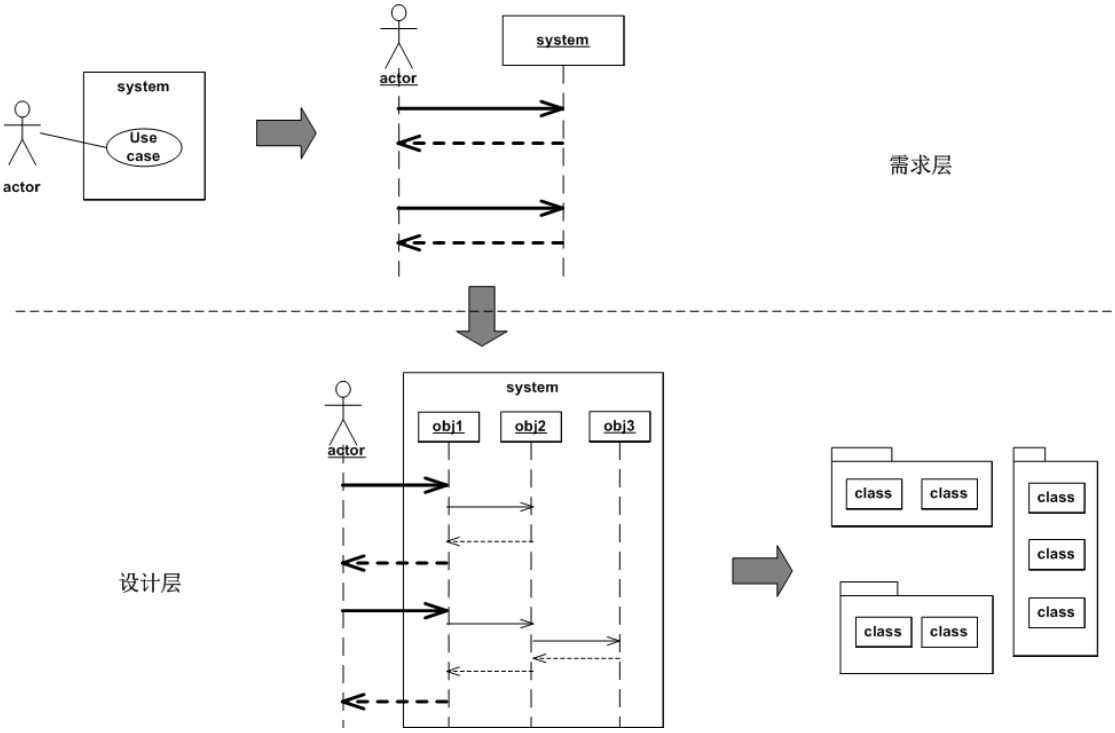


图 14-7　从用例，到用例规约，到一组对象的协作，到模块划分结构

聚焦架构设计环节，用例驱动的模块划分过程如图 14-8 所示。

■ 第 1 步：回答“实现用例需要哪些类”的问题——运用鲁棒图、序列图

■ 第 2 步：回答“这些类应划归哪些模块”的问题——运用包图等

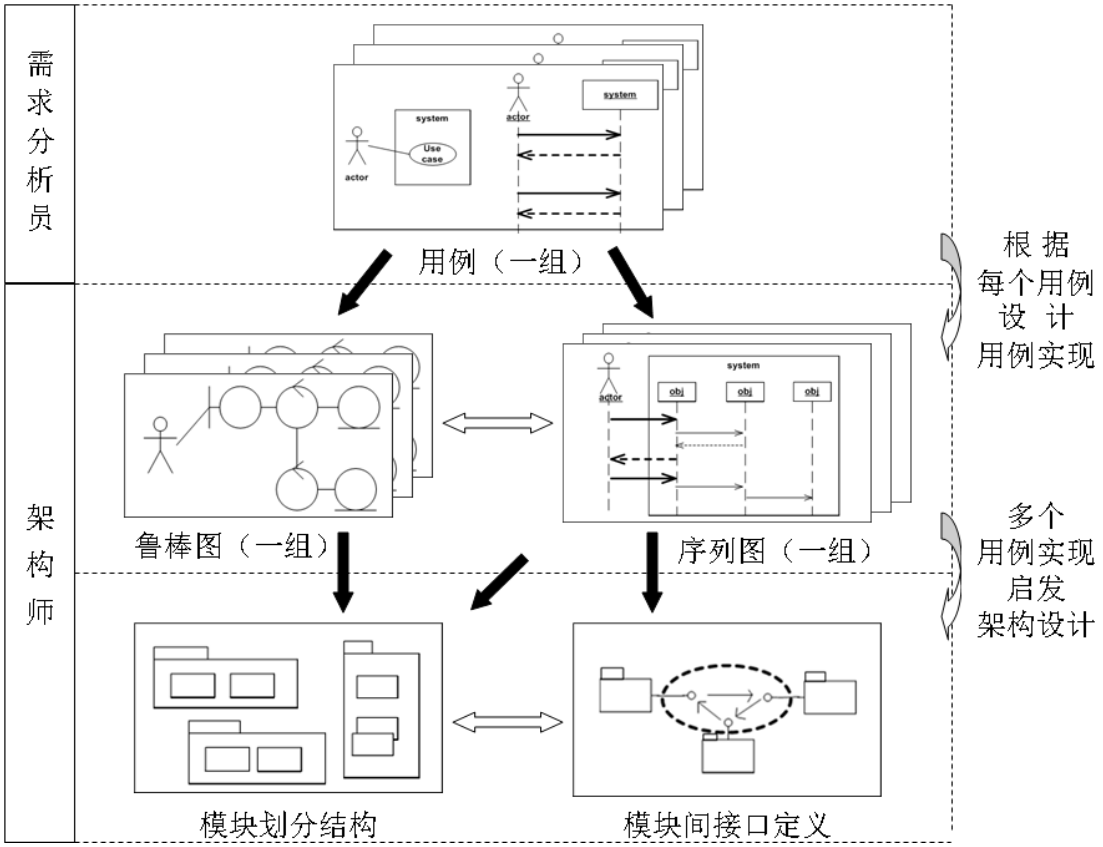


图 14-8　用例驱动的模块划分过程

14.2.2　第 1 步：实现用例需要哪些类

【技术 1】运用鲁棒图，发现实现用例需要哪些类

如果系统复杂、或者你不太熟悉领域，可以借助鲁棒图来一步步发现用例（功能）背后应该有哪些类。

例如，类似 WinZip、WinRar 这样的压缩工具大家都用过，请一起来，运用鲁棒图为其中的“压缩”功能进行设计。关键技巧是“增量建模”。

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 14-9 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 14-9　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“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 14-10 所示。

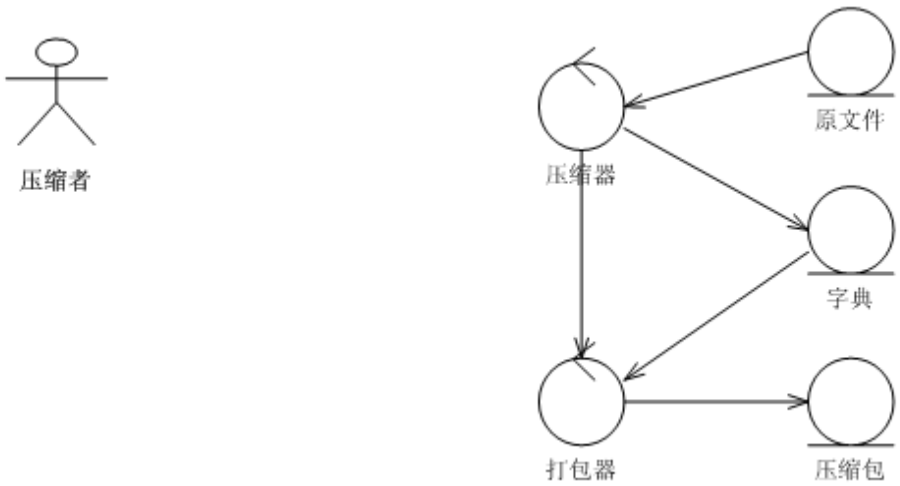


图 14-10　增量建模：开始考虑职责间关系，并发现新职责

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 14-11 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。

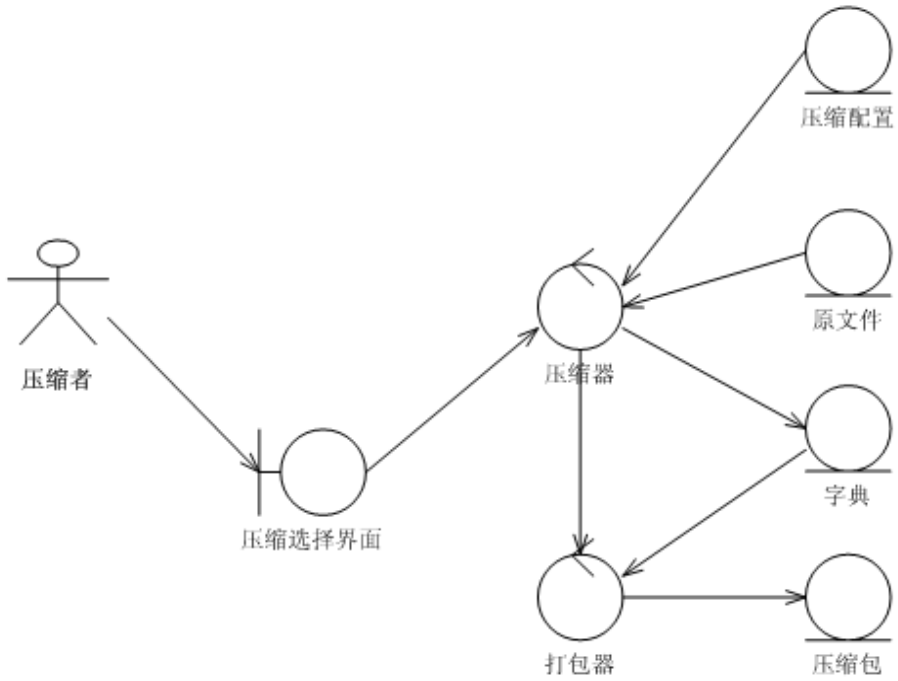


图 14-11　增量建模：继续考虑职责间关系，并发现新职责

……最终的鲁棒图如图 14-12 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。

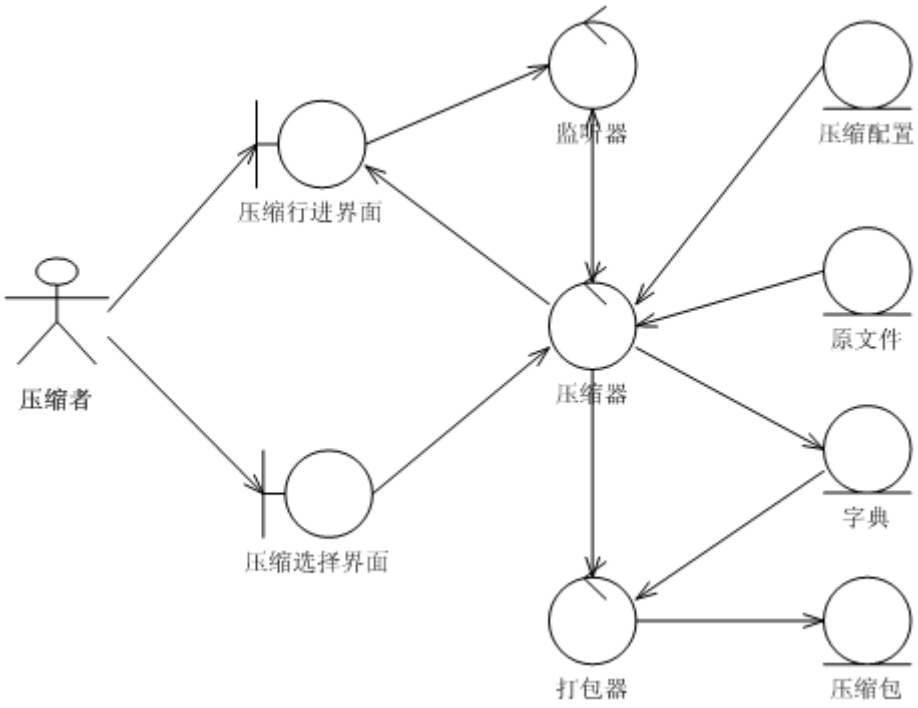


图 14-12　增量建模：直到模型比较完善

【技术 2】运用序列图，明确类之间的交互关系

UML 图“怎么画”是一件相对简单的事，确定 UML 图里“画什么”才是最难的。因为后者就是设计工作本身了。所以，如何运用 UML 图来辅助思维、启发思考？有没有一些高效技巧？就很有用了。

类似上面讲的“增量建模”，画 UML 序列图时也应注意“先大局、后细节”地进行建模。

继续设计上面的案例 WinZip。为了完成“压缩”功能，我们通过鲁棒图识别了界面（UI）、压缩器（Zipper）、打包器（Packager）、监听器（Listener）、压缩配置（ArchiveCmd）等类，图 14-13 展示了序列图建模第 1 步的设计。运用“先大局、后细节”的设计思维：

- 界面负责和用户交互，之后界面创建一个对象名为 cmd 的 ArchiveCmd 类实例；
- 界面调用压缩器的 Zip( )方法，传递参数 cmd；
- 压缩器进行循环处理，循环继续的条件是还能够成功地“Get Next File（得到下一个待压缩文件）”；
- 针对每个 file，都要进行压缩处理，其中还会涉及在内存中缓冲保存、最终写入压缩包、分卷处理等许多设计，因此不要一下展开这些设计，先封装到一个 ZipOneFile(file)方法中。

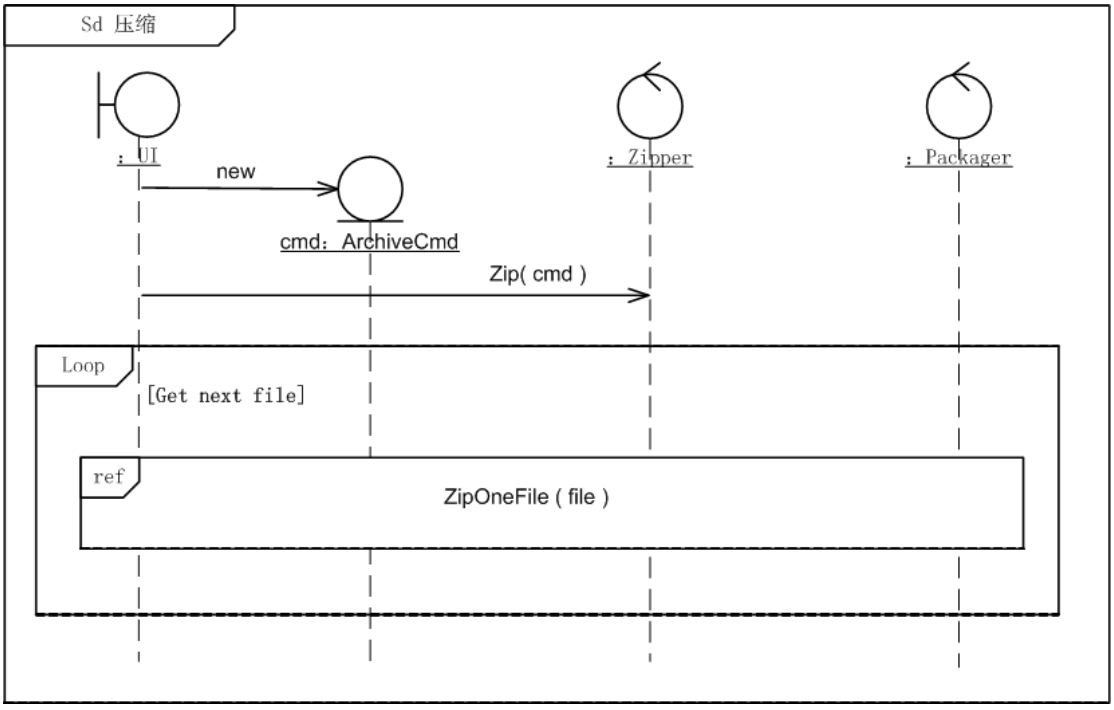


图 14-13　用例“压缩”的序列图：“先大局、后细节”建模第 1 步

这样做的好处是可以“分层次”地步步推进建模，更有利于设计思维的有效展开。如图 14-14 所示，可以专门针对 ZipOneFile( )的处理过程进行设计……在此不再详述。

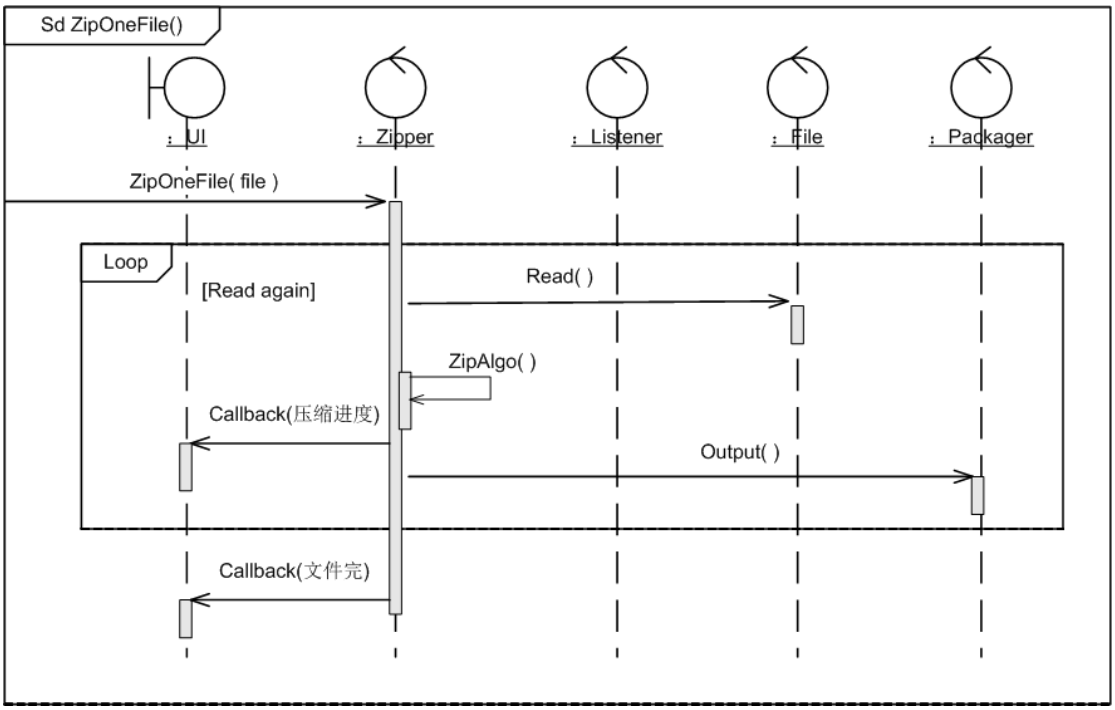


图 14-14　用例“压缩”的序列图：“先大局、后细节”建模第 2 步

【解惑 1】鲁棒图是协作图、还是类图？

鲁棒图是类图，不是协作图。

使用建模工具画图时，在“类图”或“静态结构”等能找到画鲁棒图的图元。

因为鲁棒图是类图，所以鲁棒图中的“连线”并不是严谨的对象间协作关系定义。这也是为什么“鲁棒图 + 序列图”实践方式并不重复的原因。

【解惑 2】对于大系统，用例非常多，是否要研究每个用例实现？

一方面，设计一个用例的实现，得到一组相关的类，还不足以发现整个系统的架构切分规律。

但另一方面，对于大系统，用例非常多，研究每个用例实现之后再划分模块，1）不切实际；2）有先详细设计后架构设计之嫌；3）最大的问题是，设计如何实现每个用例没有受到架构的指导和制约，可扩展高性能等整体质量属性又如何保证呢？

因此，用例驱动的设计，主要输入是一组用例。不是一个用例、也不是全部用例。对于复杂系统而言，请参考第 8 章，如何从众多用例中确定少数对架构设计关键的用例。

14.2.3　第 2 步：这些类应该划归哪些模块

识别出“一堆”类之后，再“回过头来”梳理出清晰的模块结构。

这一步，是用例驱动设计有别于所有“自顶向下”设计方法的最大不同之处、是用例驱动设计思想的精髓。

……上述类似 WinZip 的软件，识别了部分类之后，我们发现可以划分为原文件读写层、压缩包读写层、压缩控制层、界面交互层等模块。如图 14-15 所示。

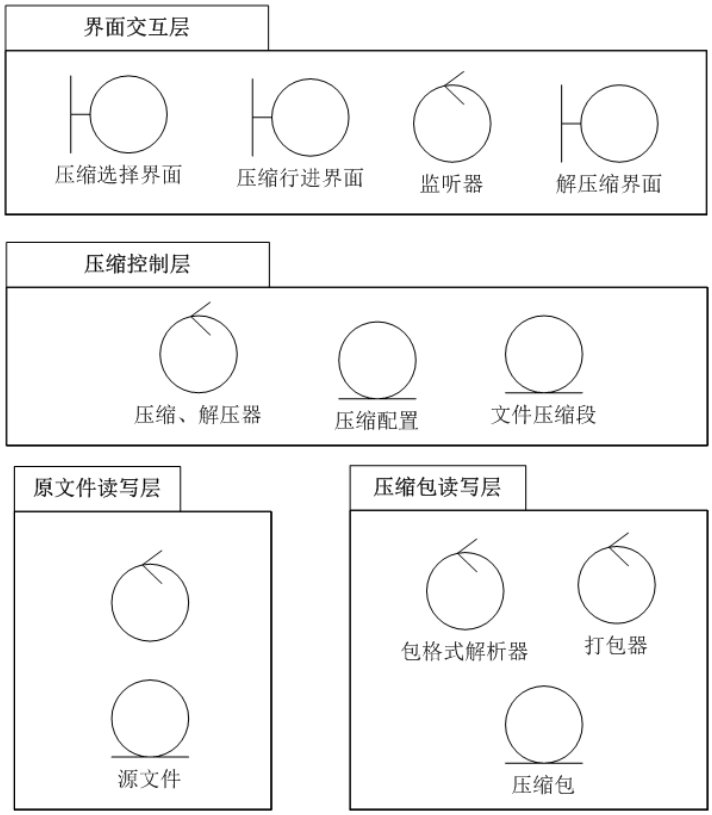


图 14-15　“从好多类，到少数模块”的设计思维过程

上述这种“从好多类，到少数模块”的设计思维过程，显然是一种“归纳”思维。自底向上思考。对比而言，第 12 章讲的“功能模块划分”、第 13 章讲的“架构分层”，都是“自顶向下”思维。这几种思维一点儿也不矛盾，可以兼收并蓄结合使用，下一章讲。

14.3　实际应用（11）——对比 MailProxy 案例的 4 种模块划分设计

14.3.1　设计

14.3.2　设计的优点、缺点