Broadview
www.broadview.com.cn

第2版

软件架构设计

程序员向架构师转型必备

温昱 著
昱培咨询 审校

经典畅销书
升级版

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

软件架构设计 第2版 程序员向架构师转型必备
電子工業出版社

软件架构设计（第 2 版）：程序员向架构师转型必备

温昱著　昱培咨询审校

電子工業出版社
Publishing House of Electronics Industry
北京・BEIJING

内 容 简 介

本书围绕“软件架构设计”主题，从“程序员”成长的视角，深入浅出地讲述了架构师的修炼之道。从“基础篇”、到“设计过程篇”、到“模块划分专题”，本书覆盖了架构设计的关键技能项，并且对于架构设计过程中可能出现的各种问题给与了解答。

本书对于有志于成为架构师的程序员们具有非常有效的指导意义，对于已经成为架构师的同行们系统化规范架构设计也是一本很好的教材。

图书在版编目（CIP）数据

软件架构设计：程序员向架构师转型必备 / 温昱著. —2 版. —北京：电子工业出版社，2012.7
ISBN 978-7-121-17087-4

Ⅰ. ①软… Ⅱ. ①温… Ⅲ. ①软件设计－教材 Ⅳ. ①TP311.5

中国版本图书馆 CIP 数据核字（2012）第 101071 号

责任编辑：孙学瑛
印　　刷：
装　　订：
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编 100036
开　　本：787×1092　1/16　印张：16　字数：341 千字
印　　次：2012 年 7 月第 1 次印刷
印　　数：　　册　　定价：　　元

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

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

服务热线：（010）88258888。

Experts Recommend

专家推荐

（以姓氏笔划为序）

与温昱先生初识于一次部门内训，金融机构应用信息技术日久，但业务发展之快仍需信息技术部门不断思索如何提供有力的技术支持，当时系统设计人员思路难成一致，故邀请先生来讲述所得，先生讲座生动有趣，案例均为实践中心得，有助于一线设计人员在低头干事之余，能够抬头看路，从架构高度理解和看待日常工作，《软件架构设计（第 2 版）》同样着眼于研发实践，不作黄钟大吕之音，而以一觞一咏畅叙分享一线设计师的感悟体会。此书值得一看，作者亦值得一晤！

——**朱晓光** 中国建设银行 北京开发中心 处长

在厦门，曾和温老师有过 4 天晚上的坐而论道，从技术到业界、从数据模型到软件重构、从职业观到心理学，彼此颇多启发。第一时间收到本书的电子版，读来流畅易懂，胜似面晤对谈。本书内容务实、技能梳理清晰，实乃软件开发者职业生涯发展的重要参考。

——**朱志** 中国建设银行 厦门开发中心总工办

基于软件架构的开发模式，作为软件开发的最佳实践之一，越来越得到各行各业的重视和关注，但遗憾的是理解其精髓和内涵的人太少。温老师作为软件架构思想的传播者和推动者，在这本书中，对程序员如何成长为优秀的架构师给出了非常具体的指导原则和实现方法，是国内不可多得的真正将软件架构思想阐述如此精准的实践指导书。作为一名软件行业的从业者，我强烈推荐给大家。

——**李哲洙 博士** 东软集团 电信事业部 网管产品与系统部部长

这本书以架构设计人员实际工作流程为线索，详细阐述了逻辑架构和物理架构视图的重要性及其在架构设计中的应用方法。此外，本书从实践的角度，给出了架构设计的三个原则和 6 大步骤，并以具体实践过程为指导，给出了架构设计从需求分析到最后的架构设计、架构验证的完整的架构设计生命周期的实践方法，对软件研发项目团队和架构师的研发实践工作具有很好的指导意义。

——**杨勇** 中兴通讯 业务研究院　平台总工

从事软件工作近十年，由软件功能模块的程序员开始，到独立负责几个软件项目的设计开发，一直对软件架构设计比较关注，有幸听了温昱老师的“软件架构设计”讲座，顿感茅塞顿开，再次阅读温老师的《软件架构设计》，对架构设计有了更深的感悟。如果你对软件架构设计感觉朦朦胧胧，温先生的《软件架构设计（第 2 版）》定能让你拨开云雾见青天。

——**杨为禄** 南京国睿安泰信科技股份有限公司 一线软件工程师

近年来，阅读了诸多系统、需求、架构类的书籍资料，温老师的几本书简明扼要，见解独到，颇多启发。“横看成岭侧成峰，远近高低各不同”，大系统架构（体系结构）包括系统组分、组分间的关系，以及演化等三要素；温老师在本书中给出了典型视角、典型模式、典型过程等实践指南。有志创造系统，赋予软件灵魂的架构师，当读此书。

——**张雪松** 中国电子科学研究院 复杂大系统研究与仿真

架构是很玄的东西，成为优秀的架构师也是大部分程序员的理想。温昱先生这本书的特点就是从程序员角度，深入浅出地讲述了架构师的修炼之道。程序员与架构师区别的最重要一点是看待事物的角度和处理方法，优秀的程序员按照本书的方法，在日常工作中一步步实践，有助于培养出架构师的能力，从而逐步成长成为架构师。架构的目标是为了沟通和交流，温先生也深刻地领悟到这一架构设计的根本目标，并将这一目标转化为方法论。架构设计不是给自己看的，而是为了与客户、领导和团队沟通，本书的重点在于架构设计实践，从用例、需求分析、概念模型、细化模型等一步步地指导如何完成架构设计，并且对于架构设计过程中可能出现的各种问题给予了解答。本书对于有志于成为架构师的程序员们具有非常有效的指导意义，对于已经成为架构师的同行们系统化规范架构设计也是一本很好的教材。

——**钱煜明** 中兴通讯 业务研究院 移动互联网总工程师

早在 2009 年的时候就读过温老师的《软件架构设计》第一版，2011 年有幸请到温老师来公司主讲“软件架构设计”，幸有当面请教的机会，温老师对软件架构独特的授课方法和深厚的功底让我如沐春风、豁然开朗，颇有几分“顿悟”之感。

五年磨一剑，如今有幸抢先拜读温老师的《软件架构设计》第二版，更是被书中内容所折服。书中融合了作者多年来在一线的实践和培训经验，深入浅出地阐释了什么是软件架构，手把手教你从客户需求入手顺畅地设计出高可用的软件架构，让你读完本书后情不自禁地感叹：“原来软件架构设计并没有那么高深莫测！”该书理论和实践并重，是一本不可多得的软件架构设计的指导书籍。

——**崔朝辉** 东软集团 技术战略与发展部 资深顾问

站得足够高，才能看得足够远。当今 IT 的架构设计思想理念已经是经过数次洗礼之后的结晶，而温昱先生抓住了这一结晶生命体的真正骨架，并深入浅出地汇集成这本书。有了这本书，

你就可以依据自己的 Project 来高效地添加血肉，构建出独特的有机生命体。

——**谌晏生** 广州从兴 电力事业部 一线软件设计师

作者介绍

wy@yupeisoft.com

温昱　资深咨询顾问，软件架构专家。软件架构思想的传播者和积极推动者，中国软件技术大会杰出贡献专家。十五年系统规划、架构设计和研发管理经验，在金融、航空、多媒体、电信、中间件平台等领域负责和参与多个大型系统的规划、设计、开发与管理。

昱培咨询
YUPEI CONSULTING

training@yupeisoft.com

昱培咨询专注于如下三个领域的咨询与培训：

- 架构设计
- 详细设计
- 设计重构

几年来，我们为近百家软企提供了卓有成效的服务。

长期一线经验的积累，更促成了 ADMEMS 架构实践体系、ARCT 设计重构方法论的形成和成熟，并已成为了我们服务品质的重要保证之一。

ADMeMS 架构设计方法体系

ARCT重构方法论

CONTENTS

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，以及合理的人才结构，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、本科与专科的比例大致是 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。

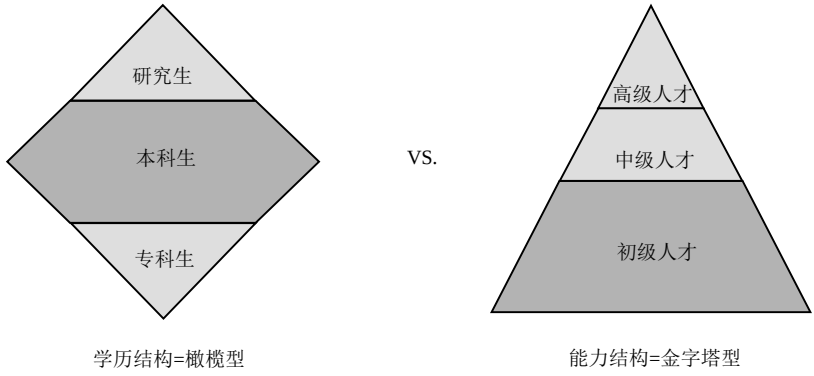


图 1-1　人才结构的两个视角

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）。

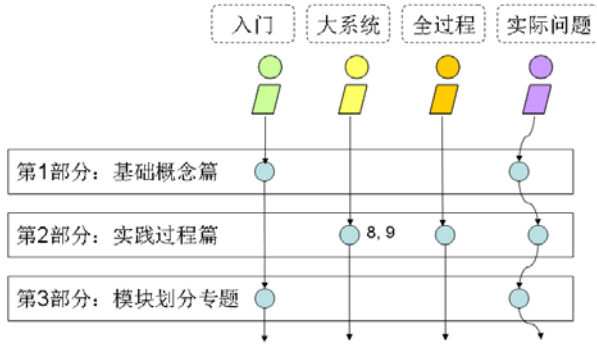


图 1-2　不同目的，不同阅读路径

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，将讨论功能模块、分层架构、用例驱动的模块划分过程等内容（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。

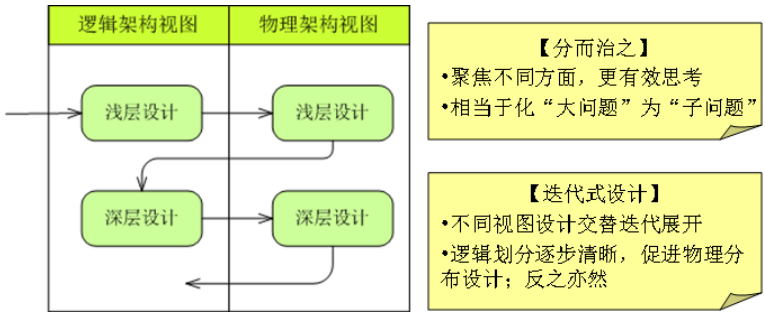


图 1-3　两视图法的“分而治之”和“迭代”思想

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。这其中，既有“水平分层”、“垂直划分功能模块”和“从用例到类、再到模块”等设计思想，也有不推荐的想到哪“切”到哪的设计方式。

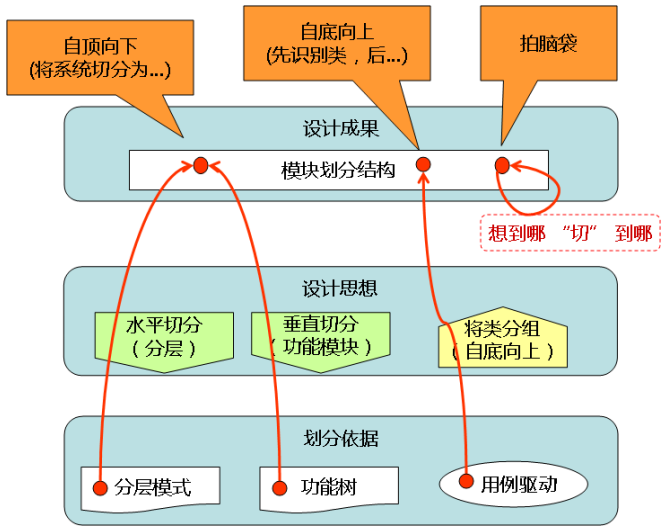


图 1-4　业界模块划分的 4 种做法

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所支撑的“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相同。

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。因此，也有人称它为“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 个选型”，如图 1-5 所示。

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）：

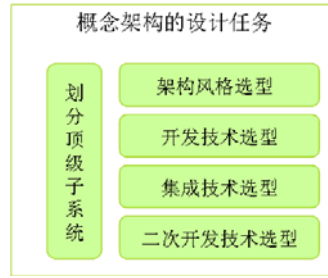


图 1-5　概念架构设计什么？

- 首先，选择架构风格、划分顶级子系统。这两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。这三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。

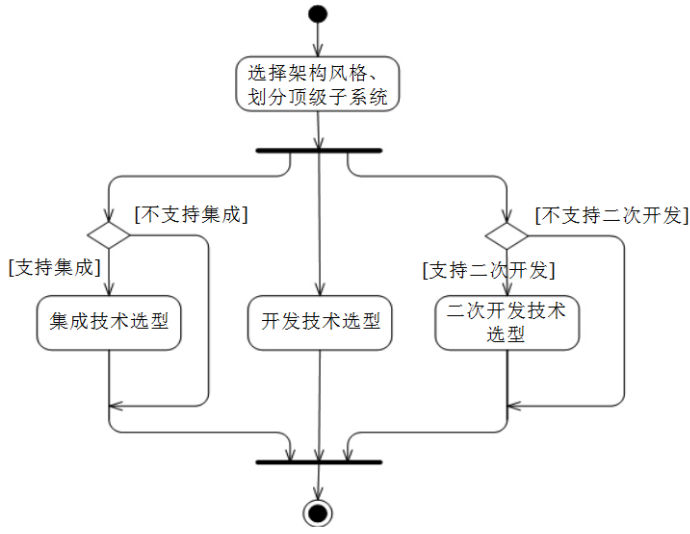


图 1-6　概念架构如何设计？

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。按如下 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——图 1-7 即，展示了从需求到架构整个过程中的关键任务项。

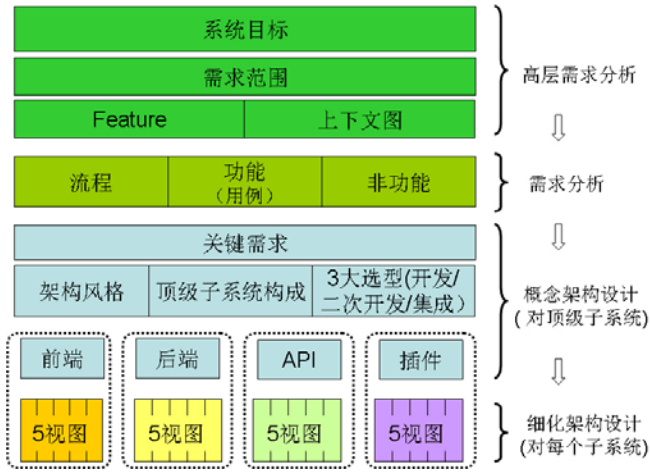


图 1-7　架构设计的全过程

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】开发人员的一个很经典的困惑是：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

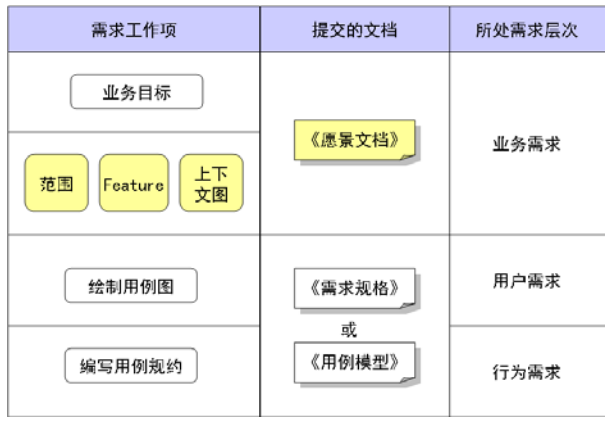


图 1-9　需求分析实践论之中型方法

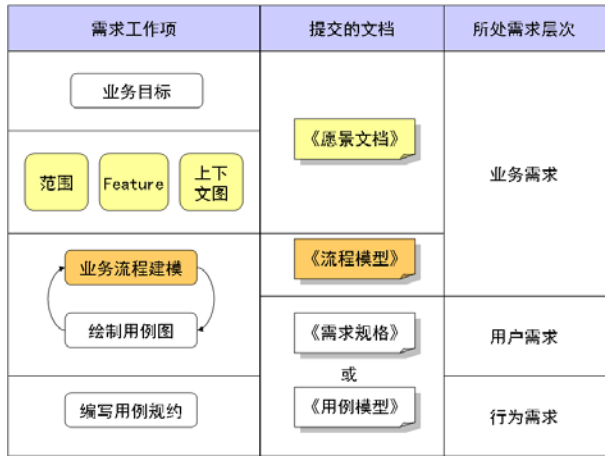


图 1-10　需求实践论之大型方法

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 2 章
解析软件架构概念

什么是架构？如果你问五个不同的人，可能会得到五种不同的答案。

——Ivar Jacobson，《AOSD 中文版》

很多人都试图给“架构”下定义，而这些定义本身却很难统一。

——Martin Fowler，《企业应用架构模式》

不积跬步，无以至千里。

程序员在向架构师转型时，都希望尽早弄清楚“什么是架构”。但是，架构的定义又多又乱，已造成“什么是架构”成了程序员向架构师转型的“大门槛”。

本章，我们讨论软件架构的概念。

值得说明的是，人们对“Architecture”有着不同的中文叫法，比如架构、构架和体系结构等。本书将一贯地采用“架构”的叫法；当然，当引用原文或提及书名时将保留原来的叫法。

2.1 软件架构概念的分类

一个词（比如“电脑”），可能并不代表一件单独的东西，而是代表了一类事物。这个一般性的表述就是我们通常所说的“概念”。

也许读者期待一个干净利落的软件架构概念，但这有点儿难。对此，Martin Fowler 给出的评价是：

> 软件业的人乐于做这样的事——找一些词汇，并将它们引申到大量微妙而又相互矛盾的含义中。一个最大的受害者就是“架构”这个词。……很多人都试图给“架构”下定义，而这些定义本身却很难统一。

本书将软件架构概念分为两大流派——组成派和决策派，帮助各级开发人员快速理清“什么是架构”的基础问题。下面就采用这种方式介绍架构概念。

2.1.1　组成派

Mary Shaw 在《软件体系结构：一门初露端倪学科的展望》中，为“软件架构”给出了非常简明的定义：

> 软件系统的架构将系统描述为计算组件及组件之间的交互。(The architecture of a software system defines that system in terms of computational components and interactions among those components.)

必须说明，上述定义中的“组件”是广泛意义上的元素之意，并不是指和 CORBA、DCOM、EJB 等相关的专有的组件概念。“计算组件”也是泛指，其实计算组件可以进一步细分为处理组件、数据组件、连接组件等。总之，“组件”可以指子系统、框架（Framework）、模块、类等不同粒度的软件单元，它们可以担负不同的计算职责。

上述定义是“组成派”软件架构概念的典型代表，有如下两个显著特点：

（1）关注架构实践中的客体——软件，以软件本身为描述对象；

（2）分析了软件的组成，即软件由承担不同计算任务的组件组成，这些组件通过相互交互完成更高层次的计算。

2.1.2　决策派

RUP（Rational Unified Process，Rational 统一过程）给出的架构的定义非常冗长，但其核心思想非常明确：软件架构是在一些重要方面所做出的决策的集合。下面看看它的定义：

> 软件架构包含了关于以下问题的重要决策：
>
> - 软件系统的组织；
> - 选择组成系统的结构元素和它们之间的接口，以及当这些元素相互协作时所体现的行为；
> - 如何组合这些元素，使它们逐渐合成为更大的子系统；
> - 用于指导这个系统组织的架构风格：这些元素以及它们的接口、协作和组合。

- **软件架构并不仅仅注重软件本身的结构和行为，还注重其他特性：使用、功能性、性能、弹性、重用、可理解性、经济和技术的限制及权衡，以及美学等。**

该定义是“决策派”软件架构概念的典型代表，有如下两个显著特点：

（1）关注架构实践中的主体——人，以人的决策为描述对象；

（2）归纳了架构决策的类型，指出架构决策不仅包括关于软件系统的组织、元素、子系统和架构风格等几类决策，还包括关于众多非功能需求的决策。

2.1.3 软件架构概念大观

下面再列举几个著名的软件架构定义，请大家：

- 结合实践，体会自己所认为的“架构”是什么，也可问问周围同事对架构的理解；
- 体会专家们给“架构”下的定义虽多，但万变不离其宗——都是围绕“组成”和“决策”两个角度定义架构的；
- 注意区分，下面的定义 1 和定义 2 属于架构概念的“决策派”，而定义 3、4、5、6、7 属于架构概念的“组成派”；
- 关注定义 7（来自 SEI 的 Len Bass 等人），它将架构的多视图“本性”体现到了定义当中，是相对比较新的定义，业界都深表认同。

1．Booch、Rumbaugh 和 Jacobson 的定义

架构是一系列重要决策的集合，这些决策与以下内容有关：软件的组织，构成系统的结构元素及其接口的选择，这些元素在相互协作中明确表现出的行为，这些结构元素和行为元素进一步组合所构成的更大规模的子系统，以及指导这一组织——包括这些元素及其接口、它们的协作和它们的组合——架构风格。

2．Woods 的观点

Eoin Woods 是这样认为的：软件架构是一系列设计决策，如果做了不正确的决策，你的项目可能最终会被取消（Software architecture is the set of design decisions which, if made incorrectly, may cause your project to be cancelled.）。

3．Garlan 和 Shaw 的定义

Garlan 和 Shaw 认为：架构包括组件（Component）、连接件（Connector）和约束（Constrain）三大要素。组件可以是一组代码（例如程序模块），也可以是独立的程序（例如数据库服务器）。连接件可以是过程调用、管道和消息等，用于表示组件之间的相互关系。“约束”一般为组件连接时的条件。

4．Perry 和 Wolf 的定义

Perry 和 Wolf 提出：软件架构是一组具有特定形式的架构元素，这些元素分为三类：负责完成数据加工的处理元素（Processing Elements）、作为被加工信息的数据元素（Data Elements）及用于把架构的不同部分组合在一起的连接元素（Connecting Elements）。

5．Boehm 的定义

Barry Boehm 和他的学生提出：软件架构包括系统组件、连接件和约束的集合，反映不同涉众需求的集合，以及原理（Rationale）的集合。其中的原理，用于说明由组件、连接件和约束所定义的系统在实现时，是如何满足不同涉众需求的。

6．IEEE 的定义

IEEE 610.12-1990 软件工程标准词汇中是这样定义架构的：架构是以组件、组件之间的关系、组件与环境之间的关系为内容的某一系统的基本组织结构，以及指导上述内容设计与演化的原理（Principle）。

7．Bass 的定义

SEI（Software Engineering Institute，SEI，美国卡内基・梅隆大学软件研究所）的 Len Bass 等人给架构的定义是：某个软件或计算机系统的软件架构是该系统的一个或多个结构，每个结构均由软件元素、这些元素的外部可见属性、这些元素之间的关系组成。（The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.）

2.2　概念思想的解析

2.2.1　软件架构关注分割与交互

架构设计是分与合的艺术。

“软件系统的架构将系统描述为计算组件及组件之间的交互”，Shaw 的这个定义从“软件组成”角度解析了软件架构的要素：组件及组件之间的交互。如图 2-1 所示（采用 UML 类图），架构=组件+交互，组件和组件之间有交互关系（图中的“交互”关系建模成 UML 关联类）。

下面以大家熟悉的 MVC 架构为例进行说明。如图 2-2 所示。

- 采用 MVC 架构的软件包含了这样 3 种组件：Model、View、Controller。
- 这 3 种组件通过交互来协作：View 创建 Controller 后，Controller 根据用户交互调用 Model 的相应服务，而 Model 会将自身的改变通知 View，View 则会读取 Model 的信

息以更新自身。

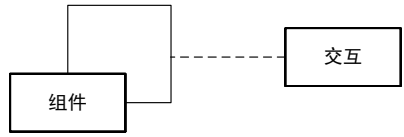


图 2-1　软件架构的要素：组件及组件之间的交互

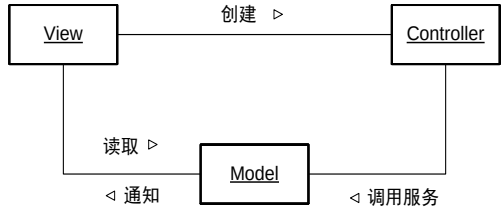


图 2-2　MVC 架构作为“组件 + 交互”的例子

通过此例可以看出，“组件＋交互”可以将 MVC 等“具体架构设计决策”高屋建瓴地抽象地表达出来。

2.2.2　软件架构是一系列有层次的决策

架构属于设计，但并非所有设计都属于架构。架构涉及的决策，往往对整体质量、并行开发、适应变化等方面有着重大影响（否则就放到详细设计环节了）：

- 模块如何划分。
- 每个模块的职责为何。
- 每个模块的接口如何定义。
- 模块间采用何种交互机制。
- 开发技术如何选型。
- 如何满足约束和质量属性的需求。
- 如何适应可能发生的变化。
- ……

还有很多

而且实际的设计往往是分层次依次展开的——无论是决策如何切分系统还是决策技术选型都是如此：

- 例如，你设计一个 C/S 系统时，是不是经历着这样一个“决策树”过程：……嗯，我决定采用 C/S 架构，系统包含 Client 和 Server；……嗯，我决定将 Server 分为三层；……嗯，我决定将 Server 的引擎层划分为 N 个模块；……（如图 2-3 所示。）
- 例如，你设计一个 B/S 系统时，可曾有过这样的“决策过程”：……嗯，我决定 B/S

前端采用 JSP 技术；……嗯，具体到 Framework 我选 Struts；……（如图 2-4 所示。）

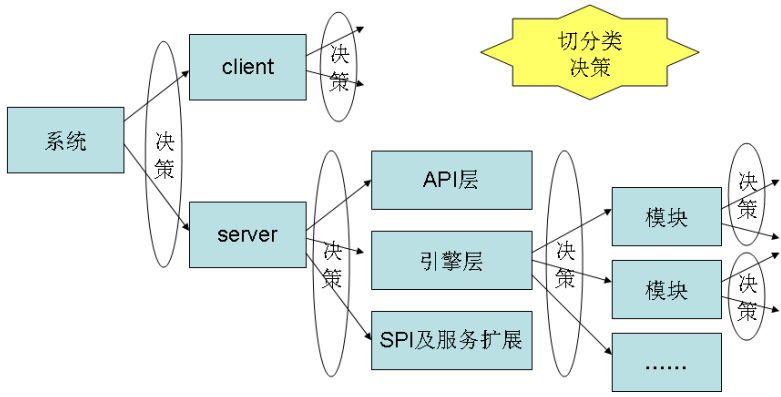


图 2-3　架构设计过程是一棵决策树（切分类决策）

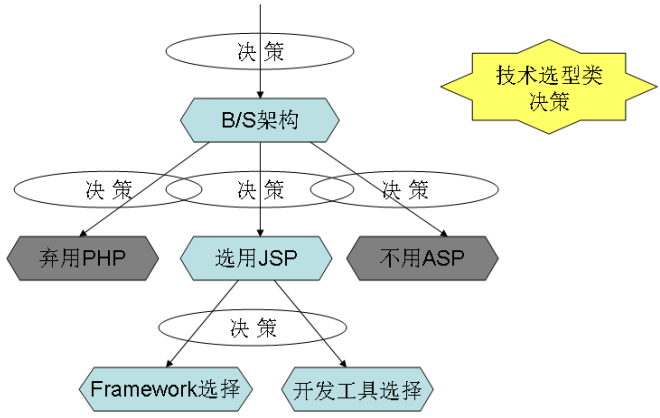


图 2-4　架构设计过程是一棵决策树（技术选型类决策）

再举一例。现在你来设计一个硬件设备调试系统。

第 1 步，理解需求——此时软件系统是黑盒子

如图 2-5 所示，你要设计的系统现在还未切分，它的主要需求目标有：

- 作为设备调试系统，其主要功能是实时显示设备状态，以及支持用户发送调试命令；
- 另外，由于是为硬件产品配套的软件系统，所以它必须容易被测试，否则是硬件故障还是软件故障将很难区分；
- 再就是必须具有很高的性能，具体性能指标为“每秒钟能够刷新 5 次设备状态的显示，并同时支持一个完整命令字的发送”。

第 2 步，首轮决策——此时软件系统被高层切分

之后，软件架构师必须规划整个系统的具体组成。通常，对于一个独立的软件系统而言，它常常被划分为不同的子系统或分系统，每个部分承担相对独立的功能，各部分之间通过特定的交互机制进行协作。

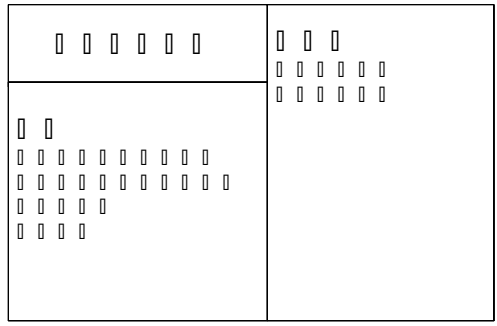


图 2-5　设备调试系统：主要目标（CRC 卡）

而此例中的设备调试系统则不同，它有两个相对独立的应用组成：一个桌面应用和一个嵌入式应用。

那么，它们如何通信呢？最终决定，将它们通过串口连接，采用 RS232 协议进行通信。

再接下来，架构师必须决定这两个应用分别担负哪些职责（如图 2-6 中的 CRC 卡所示）：桌面应用部分负责提供模拟控制台和状态显示；而嵌入式应用部分负责设备的控制和状态数据的读取。

图 2-6　设备调试系统：组成部分（CRC 卡）

第 3～*N* 步，继续决策——此时软件系统被切分成更小单元

……现在，设备调试系统的桌面应用部分，也要划分成模块吧（如图 2-7 所示）。

通信部分被分离出来作为通信层，它负责在 RS232 协议之上实现一套专用的“应用协议”：当应用层发送来包含调试指令的协议包时，它会按 RS232 协议将之传递给嵌入部分；当嵌入部分发送来原始数据时，它将之解释成应用协议包发送给应用层。

而应用层负责设备状态的显示，提供模拟控制台供用户发送调试命令，并使用通信层和嵌入部分进行交互。

……如此步步设计下去，你就该问自己“架构的哪些目标还未达成”诸如此类的问题了。例如“应用层”怎样高速响应用户？例如“通信层”如何高性能地接受串口数据而不造成数据丢失？在此不再赘述。

桌面应用::应用层	协作者 • 通讯层
职责 • 负责设备状态的显示 • 提供模拟控制台供用户发送调试命令 • 使用通讯层和嵌入部分进行交互	

桌面应用::通信层	协作者 • 嵌入式应用
职责 • 负责在RS232协议之上实现一套专用的“应用协议” • 当应用层发送来包含调试指令的协议包，负责按RS232协议将之传递给嵌入部分 • 当嵌入部分发送来原始数据，将之解释成应用协议包发送给应用层	

图 2-7　其中的桌面应用：进一步分解（CRC 卡）

2.2.3　系统、子系统、框架都可以有架构

虽然我们最常听到的说法是“软件系统的架构”，但未必是完整的软件系统才有架构。真实的软件其实是“由组件递归组合而成”的，图 2-8 运用 Composite 模式刻画了这一点：

- 组件的粒度可以很小，也可以很大；任何粒度的组件都可以组合成粒度更大的整体。即所谓的粒度多样性问题；
- 组件粒度的界定，必须在具体的实践上下文中才有意义；你的大粒度组件，对我而言可能是原子组件。即所谓的粒度相对性问题；
- 组件分为原子组件和复合组件两种；在特定的实践上下文中，原子组件是不可再分的；复合组件是由其他组件（既可以是原子组件，又可以是复合组件）组合而成的；无论是原子组件还是复合组件，它们之间都可以通过交互来完成更复杂的功能。

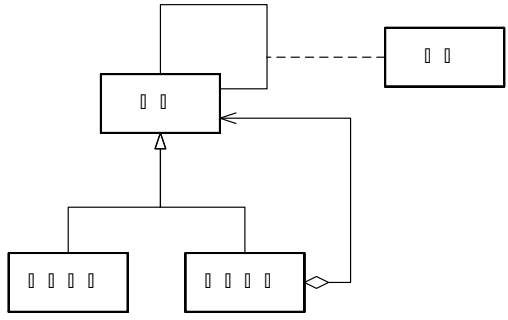


图 2-8　借助 Composite 模式刻画真实的软件

是时候为“软件架构”找准位置了，答案看上去惊人的简单（如图 2-9 所示）：

- 架构设计是针对作为复合整体的“复杂软件单元”的，架构规定了“复杂软件单元”如何被设计的重要决策；
- 实际上，系统、子系统和框架（Framework）根据需要都可以进行架构设计。

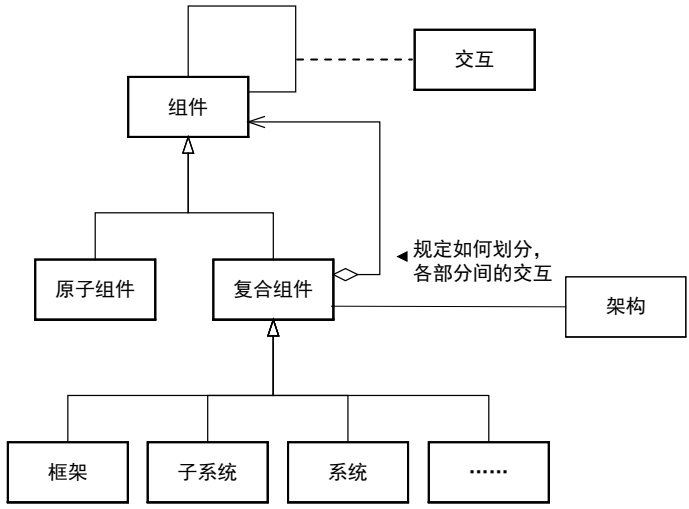


图 2-9　为“软件架构”找准位置

例如，航空航天领域的系统往往极为复杂，这样一来，总的系统需要配备系统架构师，子系统有时也会分别单独配备架构师。

又例如，用友华表 Cell 组件是标准的报表处理 ActiveX 组件，它提供几百个编程接口。虽然在用户看来它只是小小的“开盒即用”的 ActiveX 组件（上面说的“原子组件”），但它的研发团队从需求分析到架构设计到程序开发……都样样经历（上面说的“复杂软件单元”）。

再例如，随着面向服务架构（Service Oriented Architecture，SOA）被越来越多的人所接受，基于组件的软件工程（Component Based Software Engineering，CBSE）也为更多的人所认识。在此种情况下，整个系统的架构模式是 SOA，而每个组件本身也有自己的架构设计——在实践中不了解这一点会很危险。

推广开去，其实任何作为复合整体的复杂事物都可能有架构，比如一本书、一幢建筑物。那本“永不褪色的经典”《如何阅读一本书》中就说：“每一本书的封面之下都有一套自己的骨架（Every book has a skeleton hidden between its boards）。”

2.3　实际应用（1）——团队对架构看法不一怎么办

2.3.1　结合手上的实际工作来理解架构的含义

你们公司，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？

看法影响做法。先说程序员。如果程序员们对他们公司架构师的做法“不以为然”，会不会不按照架构进行后续的详细设计和编程呢？……这让我们不禁想到，程序代码实际体现的设计和《架构设计文档》差别很大，是很多公司都存在的现象。这背后，难道没有“程序员和架构师对架构有不同的理解”这一原因吗！

再说架构师。如果一个架构师认为“架构就是通用模块”，那么他可能就不会关心非通用单元的设计。如果一个架构师认为“架构就是技术选型”，那么他在拍板儿选择“Spring + Struts”之后就理所当然地无事可做了。如果一个架构师认为“投标时讲的架构就是架构的全部”，那么他才不管那“三页幻灯”对程序员的实际开发指导不够呢（因为没有意识到）。

怎么办呢？

一通理论说教是不行的。太空洞，到了实际工作中，理解依然、做法照旧。

结合手上的实际工作来理解架构的含义，是笔者推荐的做法。

2.3.2 这样理解“架构”对吗

某公司，研发二部，他们一直在做的软件产品叫 PM Suite，是一套项目管理系统。

老王，研发二部部长，经验丰富、设计功力也非常深厚。

小张，众多程序员的代表，开发骨干，功底扎实、头脑灵活、工作努力。小张希望能在 1~2 年之内，从程序员转型成为架构师。

关于“什么是架构”，小张是这么理解的：

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

当你发现越来越灵活地使用接口时，那么你就从程序员升级为架构师了。

老王用 3 个公式概括了一下小张的理解：

程　　序 = 类 + 类级接口

架构设计 = 定义类级接口

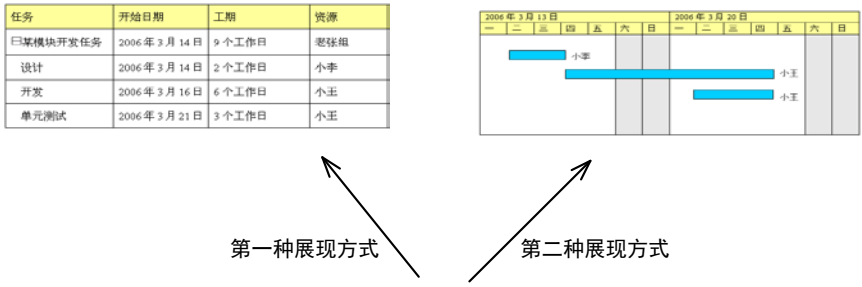
编程开发 = 像做填空题似的写实现类

这可行吗？将架构设计一下子细致到类：一是难度太大不现实，二是工作量太大没必要，三是并发呀部署呀都没考虑、性能安全可伸缩等需求能达标？……于是老王开始担心了。

2.3.3　工作中找答案：先看部分设计

老王找来小张，铺开纸笔，开始拿实际工作来“说事儿”。他说：

咱们的 PM Suite 系统，有一项名为“查看甘特图”的需求，用户要求“能够以甘特图方式查看任务的起始时间、结束时间、任务承担者等信息”。经过分析我们不难发现，PM Suite 至少应提供两种查看任务计划的方式：一种是以表格的方式将任务名称、开始时间等信息列出，另一种是采用甘特图。如图 2-10 所示。



任务如何计划，如何分配

图 2-10　PM Suite 至少要提供两种查看任务计划的方式

任务是如何计划的？又具体分配给哪些项目成员？这些信息和 PM Suite 采用何种方式来展现应当是没有关系的。根据此分析，我们立即想到采用 MVC 架构，将业务逻辑和展现逻辑分开，如图 2-11 所示。

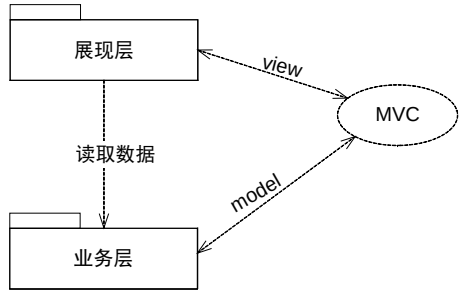


图 2-11　和具体技术无关的架构方案

上面的架构设计，还处于“和具体技术无关”的层面。我们必须考虑更多的实际开发中要涉及的技术问题，从而不断细化架构方案，这样才能为开发人员提供更多的指导和限制，也才能真正降低后续开发中的重大技术风险。

对于 PM Suite 要显示甘特图而言，“甘特图绘制包”是自行开发的，还是采用的第三方 SDK，就是一个很重要的技术问题。考虑如下：

- 一方面，用户根本不关心“甘特图绘制包”的问题，他们只在乎自己的需求是否得到了满足；而项目工期是很紧的，自行开发“甘特图绘制包”势必增加其他工作的压力，为什么不采用第三方 SDK 呢？
- 另一方面，短期内决定采用的第三方“甘特图绘制包”可能并不是最优的，所以并不希望 PM Suite“绑死”在特定“甘特图绘制包”上。

基于以上分析，架构师会决定：采用第三方 SDK，但会自主定义“甘特图绘制接口”将 SDK 隔离。如图 2-12 所示。

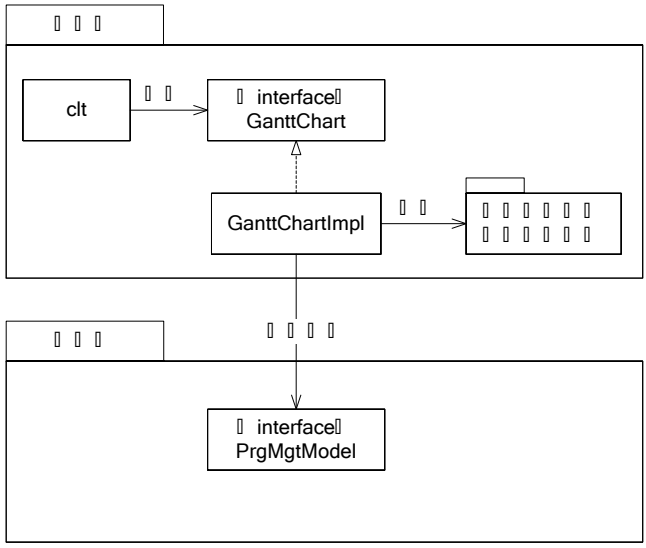


图 2-12　和技术相关的架构方案

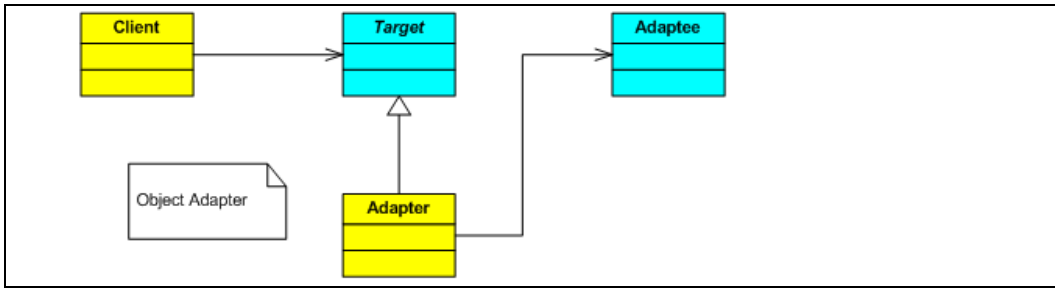
有的读者应该已经看出来了，上述设计中采用了 Adapter 设计模式。

适配器（Adapter）模式

关键字：已存在/不可预见　复用

支持变化：由于 Adapter 提供了一层“间接”，使得我们可以复用一个接口不符合我们需求的已存在的类，也可以使一个类（Adaptee）在发生不可预见的变化时，仅仅影响 Adapter 而不影响 Adapter 的客户类。

结构：


Client
Target
Adaptee
Object Adapter
Adapter

2.3.4 工作中找答案：反观架构概念的体现

看到小张听得挺有感觉，老王话锋一转，话题落于架构的理解上。他说：

有关 PM Suite 的设计，先讲到这儿，我再简单说一下“什么叫架构”。架构的定义很乱，但可以分为两派。

组成派： 架构 = 组件 + 交互

决策派： 架构 = 一组重要决策

刚才有关 PM Suite 的设计虽然仅涉及架构的“冰山一角”，但仍然可以体现软件架构的概念——对组成派和决策派的架构概念都有体现。

先说组成派的架构概念，它强调软件架构包含了“计算组件及组件之间的交互”。组件体现在哪里呢？

- **【回顾图 2-11】**。“业务层”和“展现层”就是两个组件；当然，这两个组件粒度很粗，并且完全是黑盒。
- **【到了图 2-12】**。为了支持 MVC 协作机制，设计中引入了 PrgMgtModel、GanttChart 和 GanttChartImpl 等关键类。可以说，“业务层”和“展现层”两个组件在某种程度上已从黑盒变成了灰盒，从而能提供更具体的开发指导。

那么，软件架构概念中所说的“交互”体现在哪里呢？

- **【回顾图 2-11】**。“业务层”和“展现层”两个粗粒度组件之间的交互为：展现层从业务层“读取数据”。
- **【到了图 2-12】**。“读取数据”这一交互已经“具体落实”成了“GanttChartImpl 从 PrgMgtModel 读取数据”。另外，图中还有两个“调用”关系。

由此看来，组成派软件架构概念完全是对架构设计方案的忠实概括，只不过有一点儿抽象罢了。

再看看决策派的架构概念，它归纳了架构决策的类型，指出架构决策不仅包括关于软件系统的组织、元素、子系统、架构风格等的几类决策，还包括关于众多非功能需求的决策。

- **【回顾图 2-11】**。业务层和展现层分离，体现了架构概念中的“软件系统的组织”决策，业界也普遍认同。
- **【到了图 2-12】**。为了防止 PM Suite“绑死”在特定甘特图绘制包上，设计中引入自主定义的 GanttChart 接口，让实现该接口的 GanttChartImpl 转而调用第三方 SDK（其实就是 Adapter 设计模式）。这样一来，架构就有了弹性——当发现功能更强大的甘特图

程序包时（或决定直接调用 Java 2D 自行开发甘特图绘制部分时），可以方便地仅更改 GanttChartImpl，而其他组件不用更改（如图 2-13 所示）。

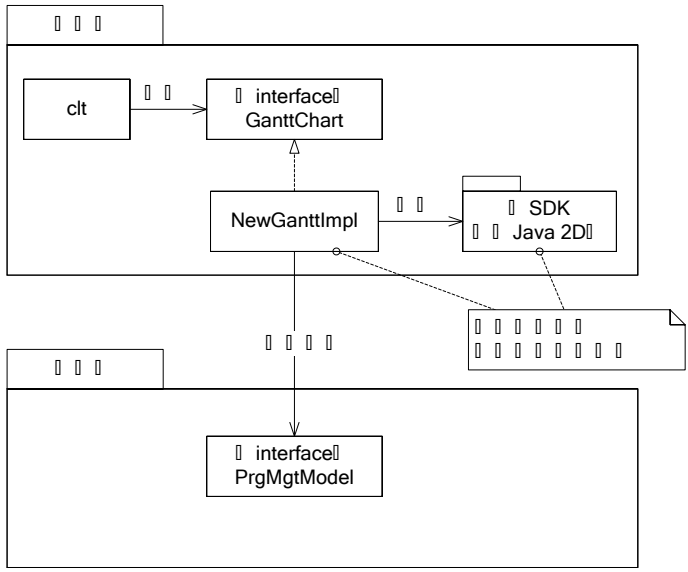


图 2-13　软件架构如何具有弹性

老王最后说：呵呵，组成派和决策派无非是个叫法，它们只不过是所站的角度不同罢了，你在具体设计一个架构时都会有所体现的。

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》

架构被用做销售手段，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）通信协议设计制定、跟踪研究；……
- 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 概念架构是什么

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、网络通信、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（MW）、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）。

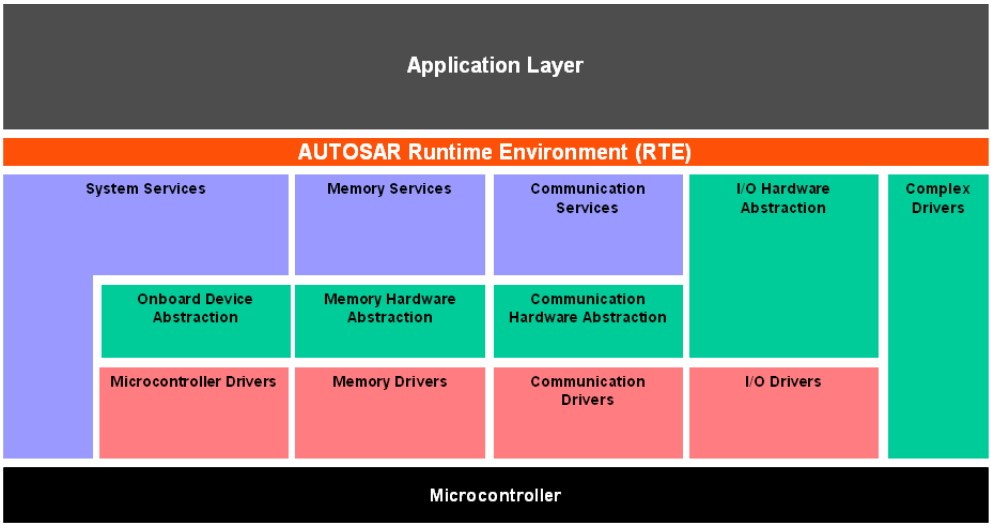


图 9-1　AUTOSAR 软件架构层次图

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、汽车网络通信和管理服务、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。

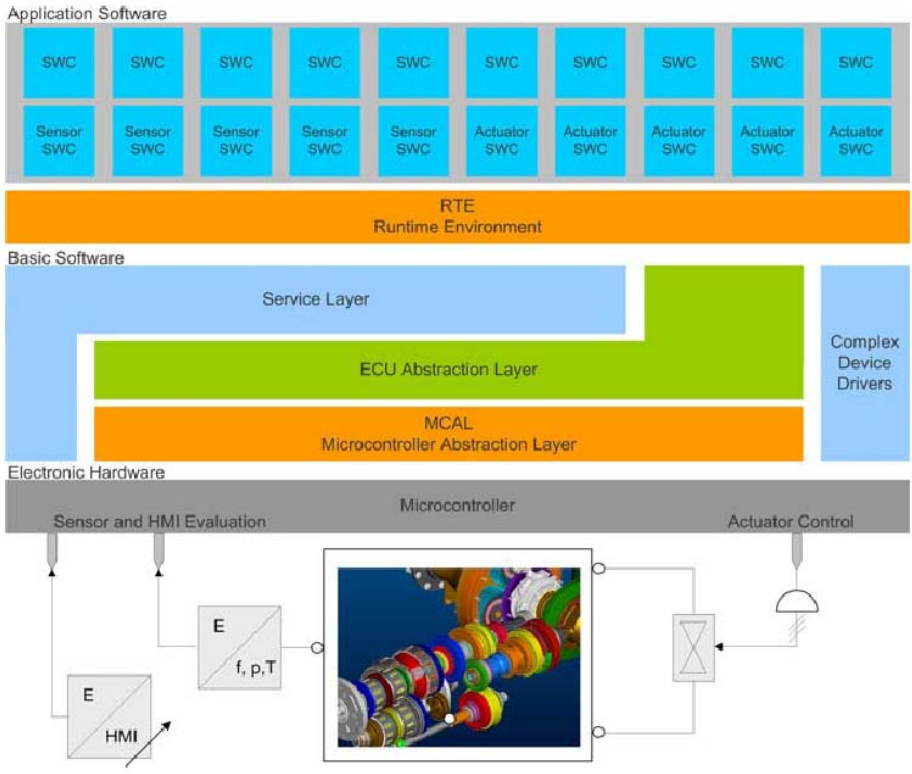


图 9-2　AUTOSAR 软件架构层次图

关键设计思想之二是 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。

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。

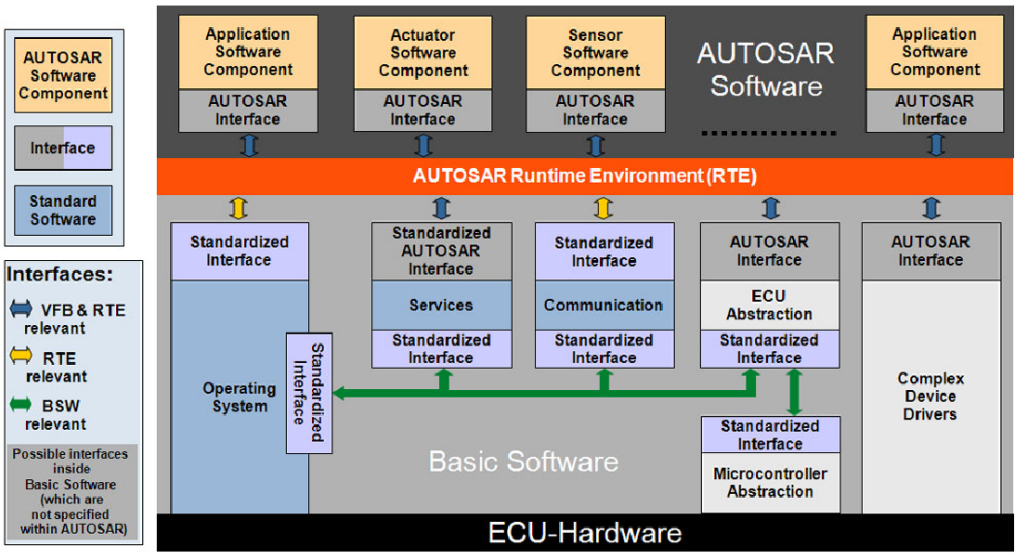


图 9-3　RTE 是 AUTOSAR 的关键设计

第三，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。

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）。

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。）

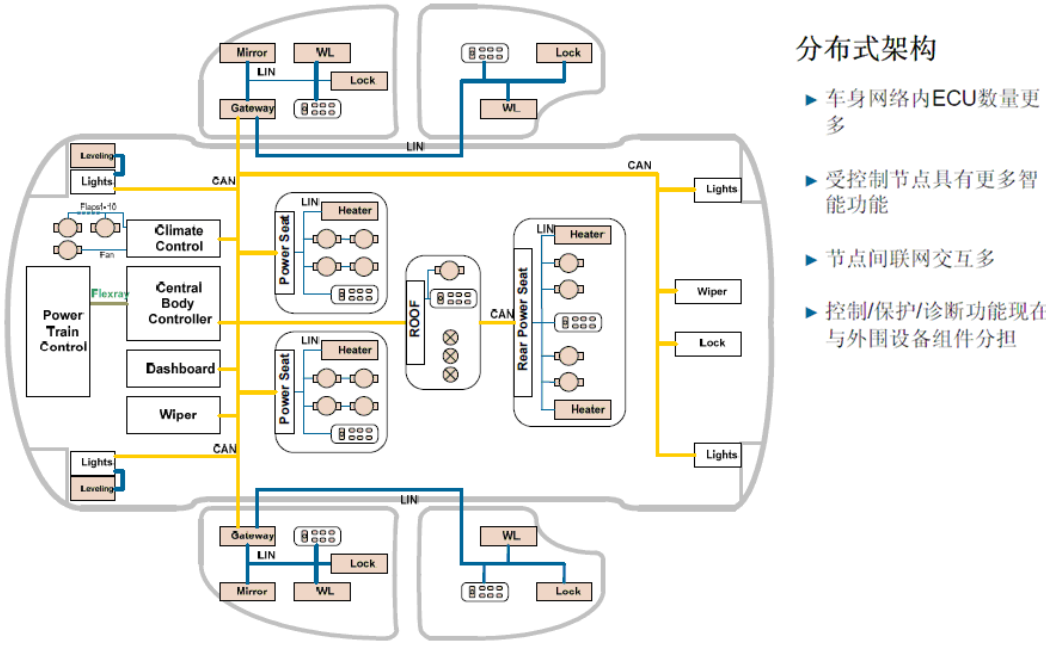


图 9-4　汽车电子的分布式架构（来源：飞思卡尔技术资料）

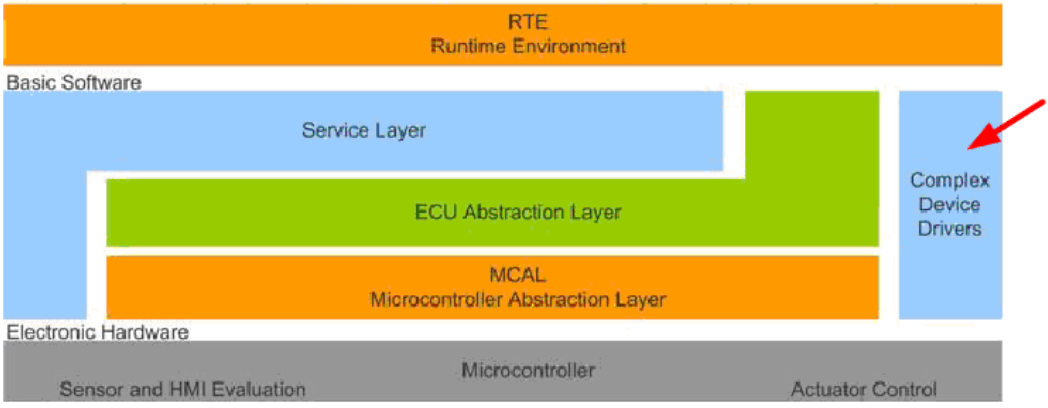


图 9-5　复杂设备驱动：放弃“层次化”，选择“一体化”

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。

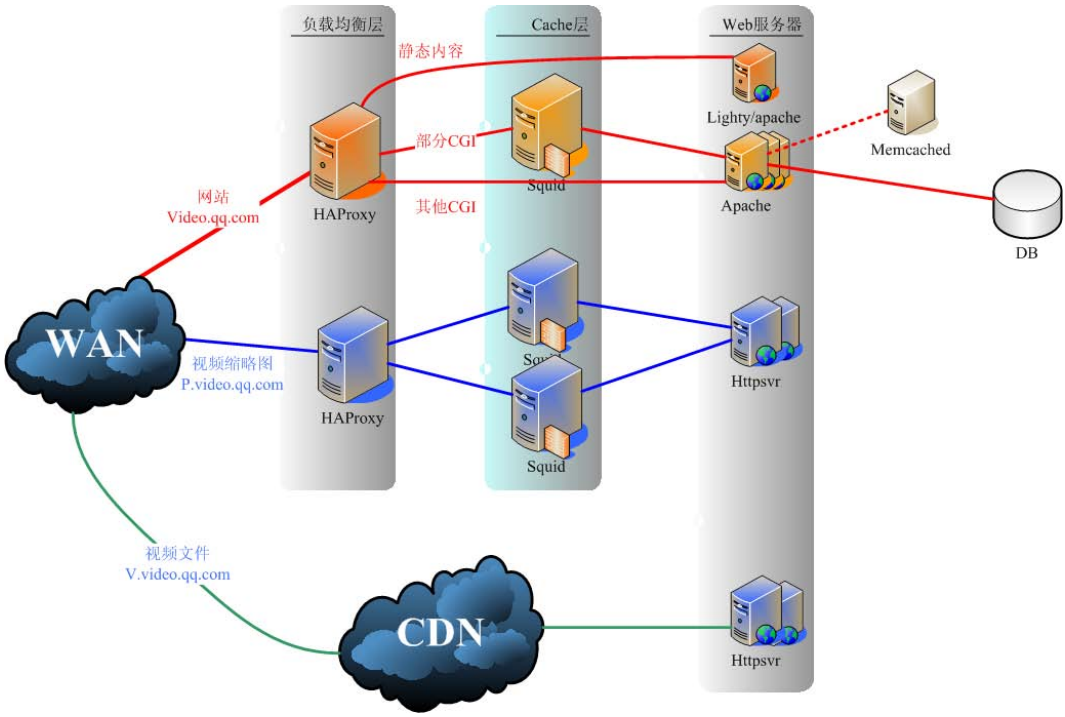


图 9-6　QQvideo Web 架构图（图片来源：腾讯大讲堂 http://djt.open.qq.com）

既然是概念架构举例，我们就“抠”一下定义（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。
  - 针对视频、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。

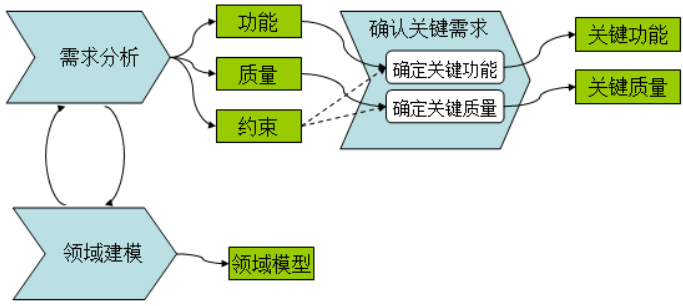


图 9-8　需求分析、领域建模和确定关键需求

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）。

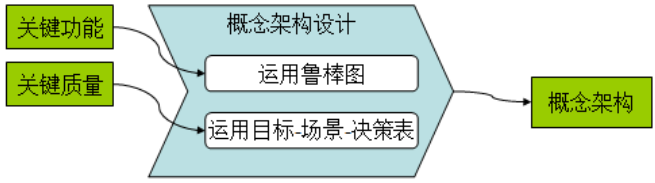


图 9-9 “关键需求”进 ,“概念架构”出

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”的错误。

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，这一点恰恰是必需的。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；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”在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：

- 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”就是下面一节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，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”正是因为后者检查的作用，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。

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 9-10　鲁棒图的元素

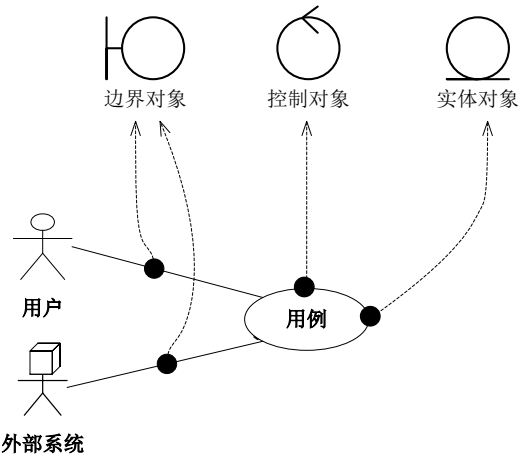


图 9-11　研究用例，进行鲁棒图建模

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）。

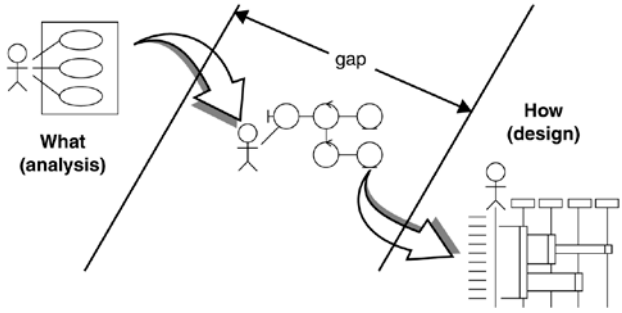


图 9-12　鲁棒图的“桥梁”作用（图片来源：《UML 用例驱动对象建模》）

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；

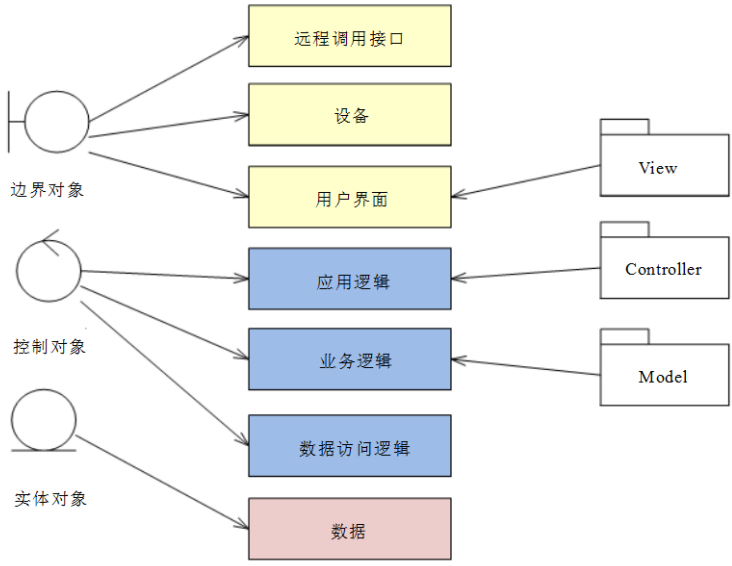


图 9-13　鲁棒图 3 元素的具体化，以及与 MVC 的对比

- 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。

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。

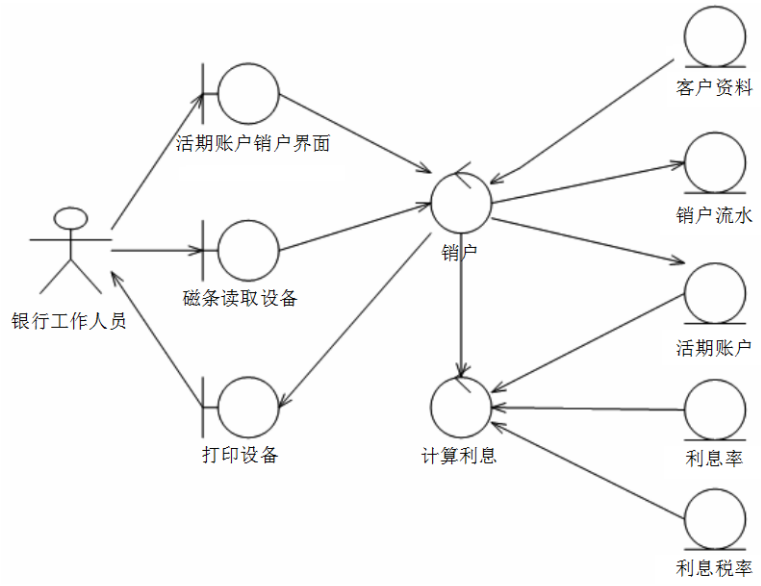


图 9-14 “销户”的鲁棒图

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。

举网上书店系统这个例子，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”功能运用增量建模技巧进行鲁棒图建模。

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 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。

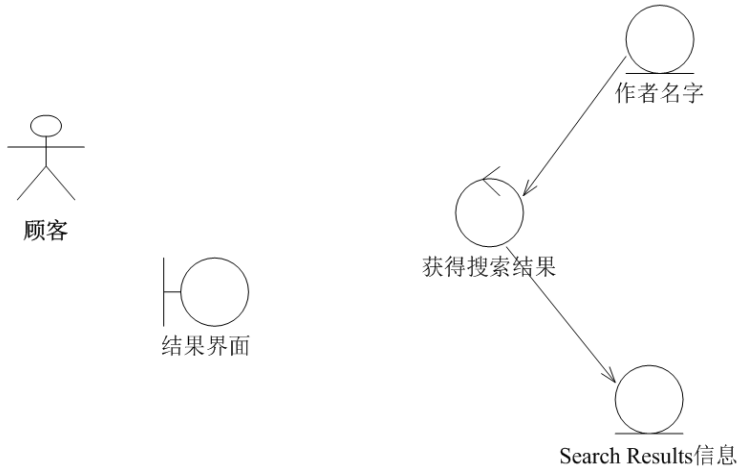


图 9-15　增量建模：先识别最“明显”的职责

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。

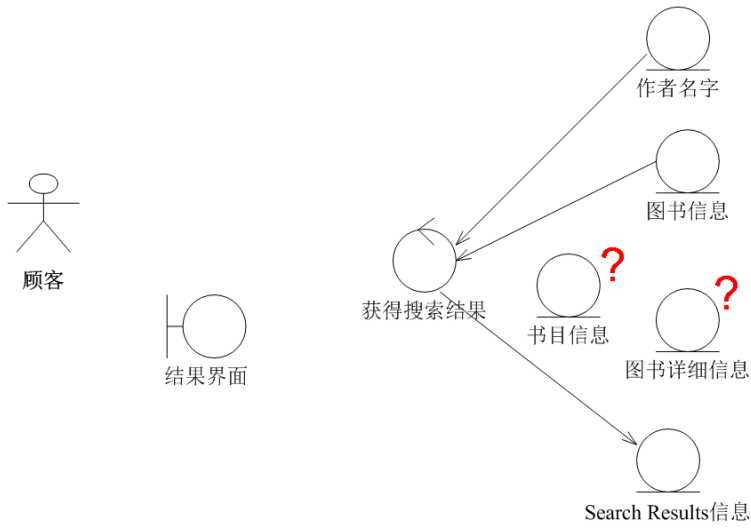


图 9-16　增量建模：开始考虑职责间关系，并发现新职责

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。

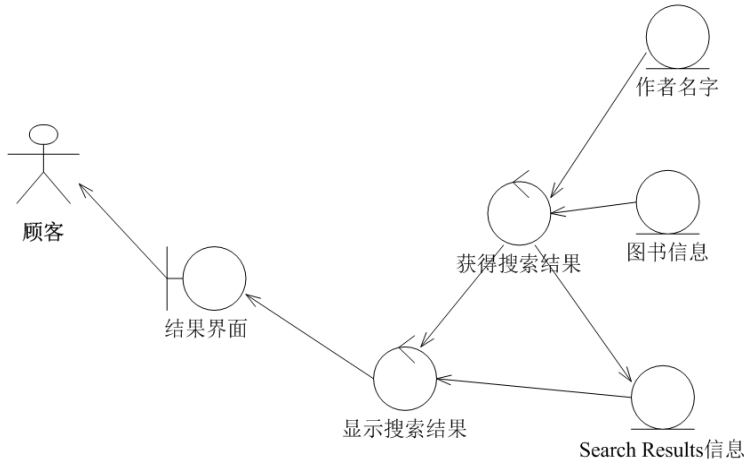


图 9-17　增量建模：继续考虑职责间关系，并发现新职责

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来驱动的，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，关键是你的头脑中要有。“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。

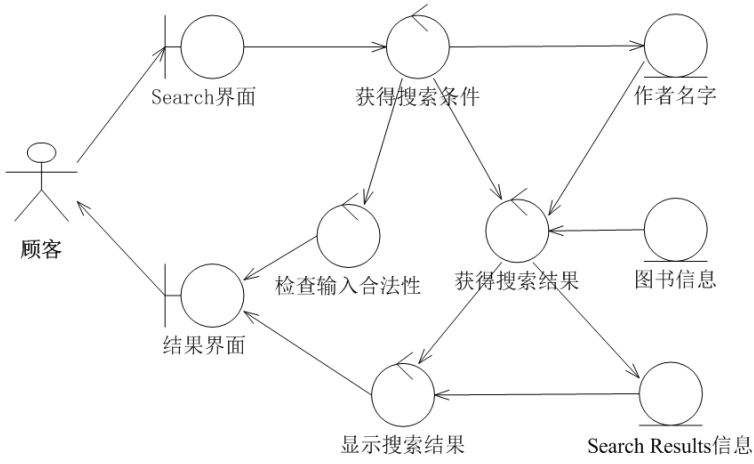


图 9-18　增量建模：直到模型比较完善

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　再谈什么样的鸿沟，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。

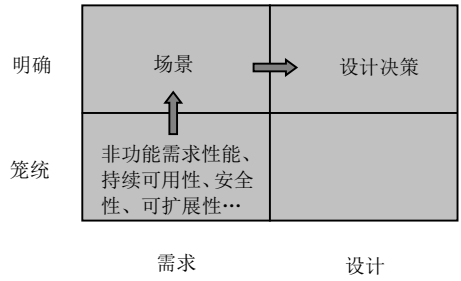


图 9-19　以场景为“跳板”的非功能目标设计思维

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怎样。（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。由此可知，工具的价值所在。

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。

场景卡 提交者：张三			
If		Then	
程序	大量更新数据	数据复制的开销	非常大
Context:已部署了多个 DBMS 实例以增加数据处理能力			

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、可视化。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）。

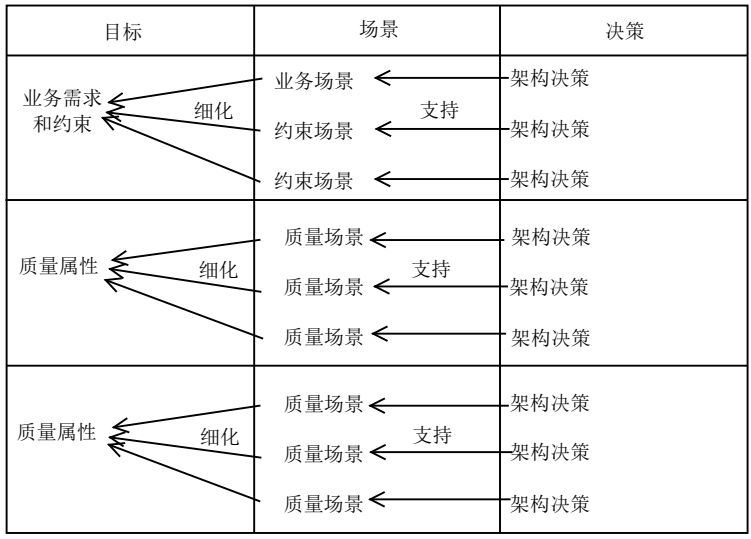


图 9-21　目标—场景—决策表

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。

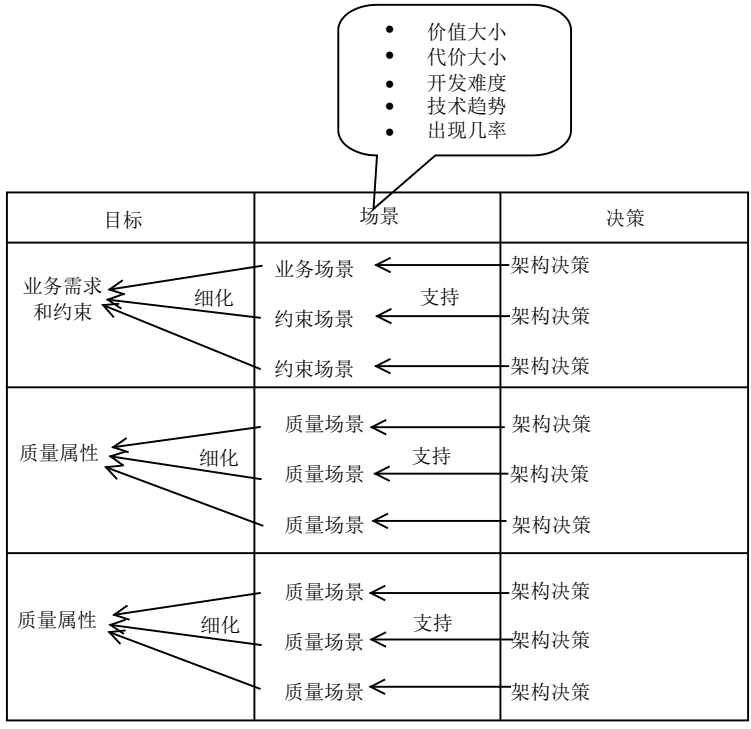


图 9-22　场景评估要考虑的一些因素

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。

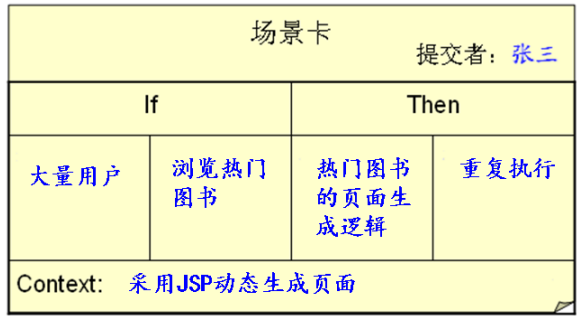


图 9-23　设计网上书店系统：场景卡

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）

续表

目标	场　景		决　策
		【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）。

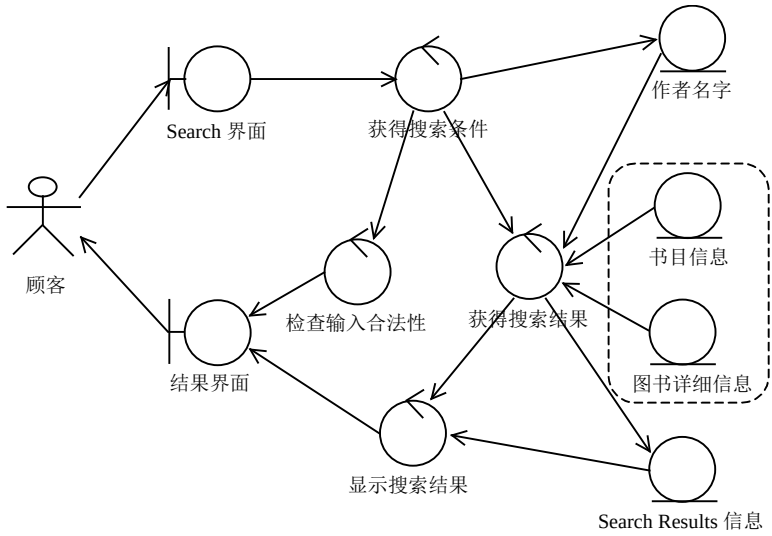


图 9-24　设计网上书店系统：更新设计

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。

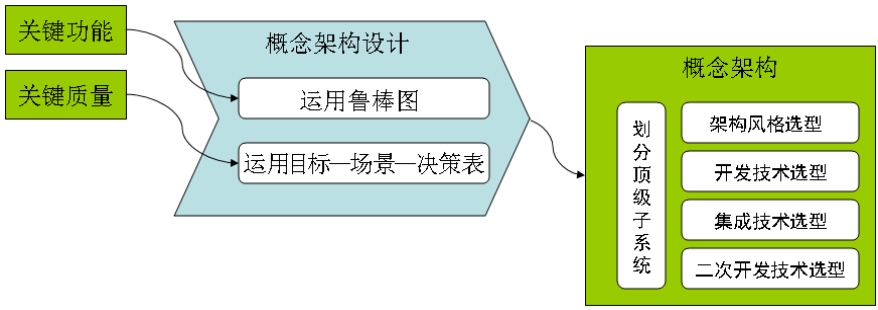


图 9-25　概念架构设计要明确的是“1 个决定、4 个选择”

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。

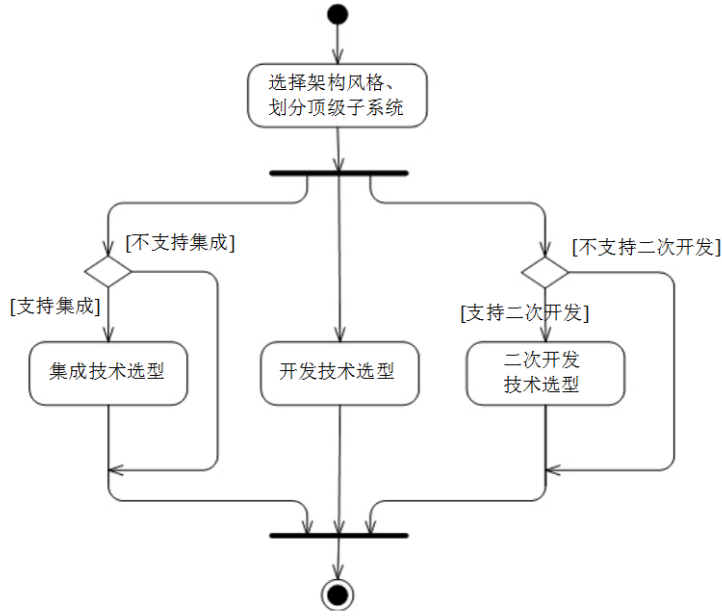


图 9-26　概念架构 5 项设计任务应该以什么顺序完成

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。表 9-2 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。下面将实际应用到 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。

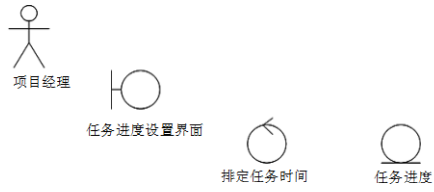


图 9-27 “制定进度计划”的增量建模过程

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”）。

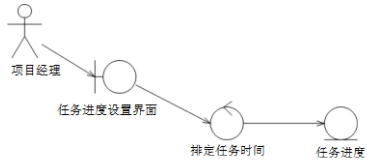


图 9-28 “制定进度计划”的增量建模过程

- 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”。

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。

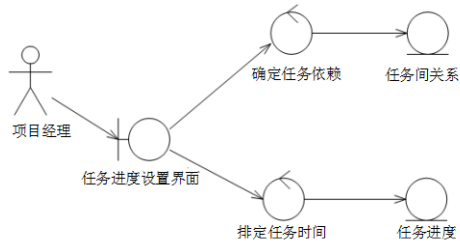


图 9-29 “制定进度计划”的增量建模过程

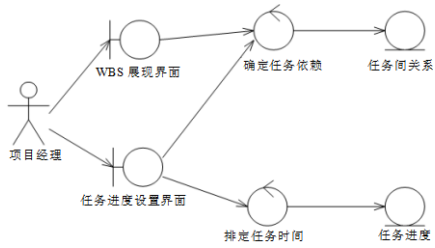


图 9-30 “制定进度计划”的增量建模过程

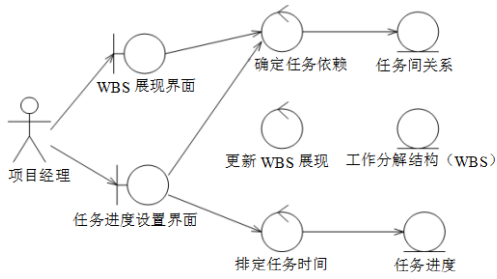


图 9-31 “制定进度计划”的增量建模过程

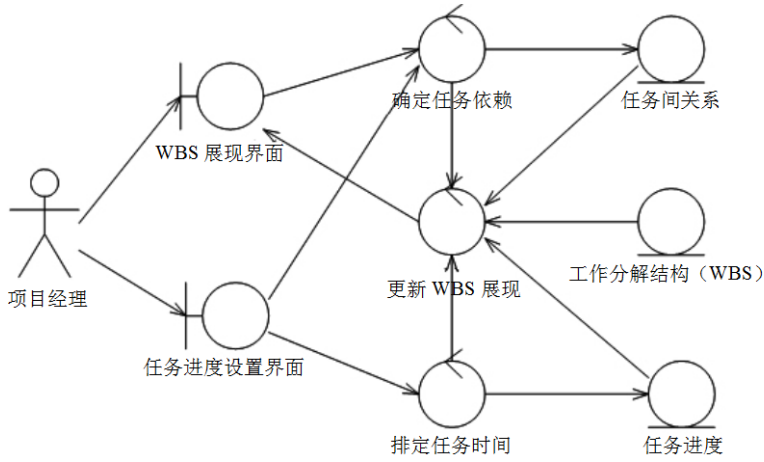


图 9-32 “制定进度计划”的增量建模过程

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。

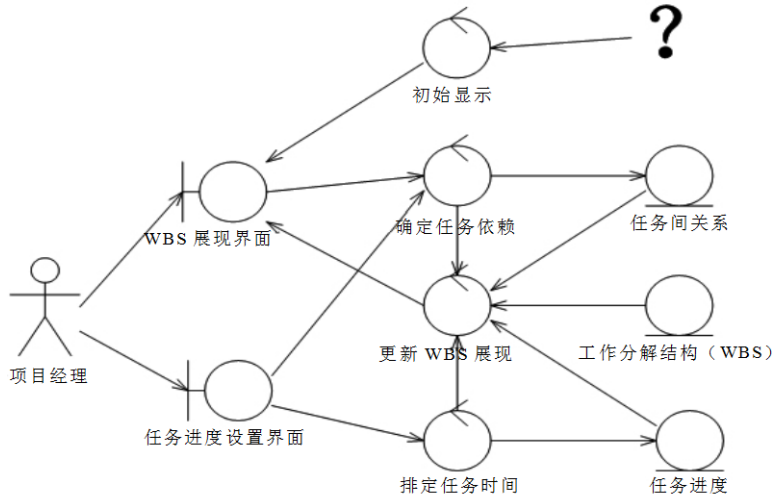


图 9-33　鲁棒图建模能启发架构级的“关键决策点”

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……

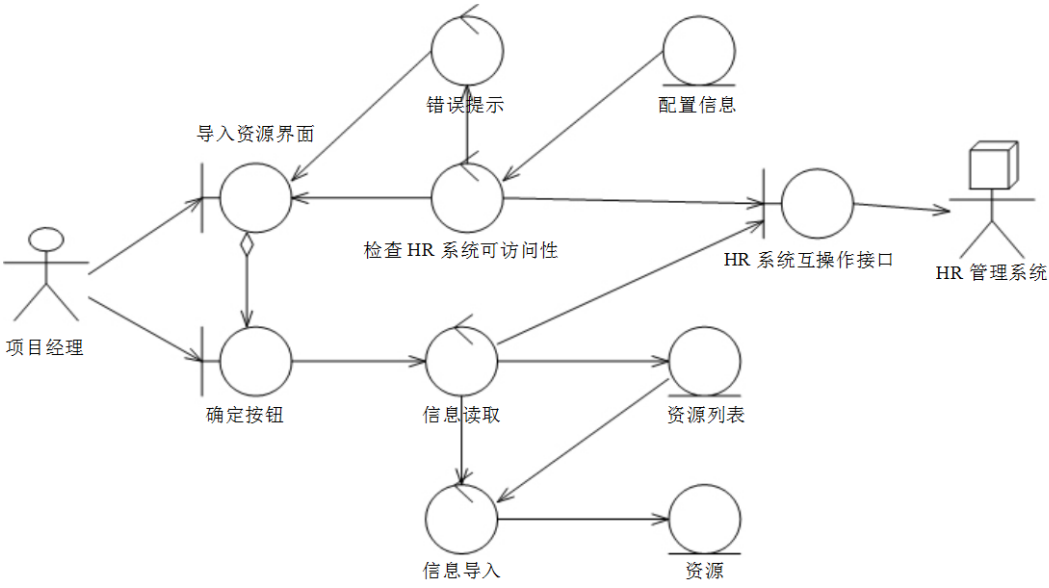


图 9-34 “从 HR 系统导入资源”的鲁棒图

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。

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

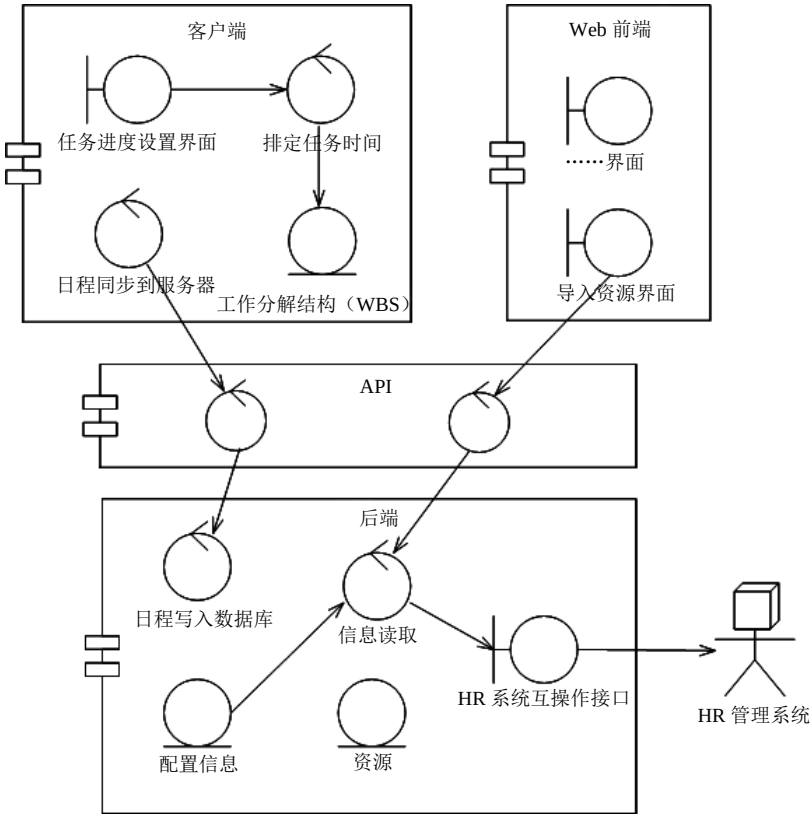


图 9-35　选择架构风格，划分顶级子系统

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。

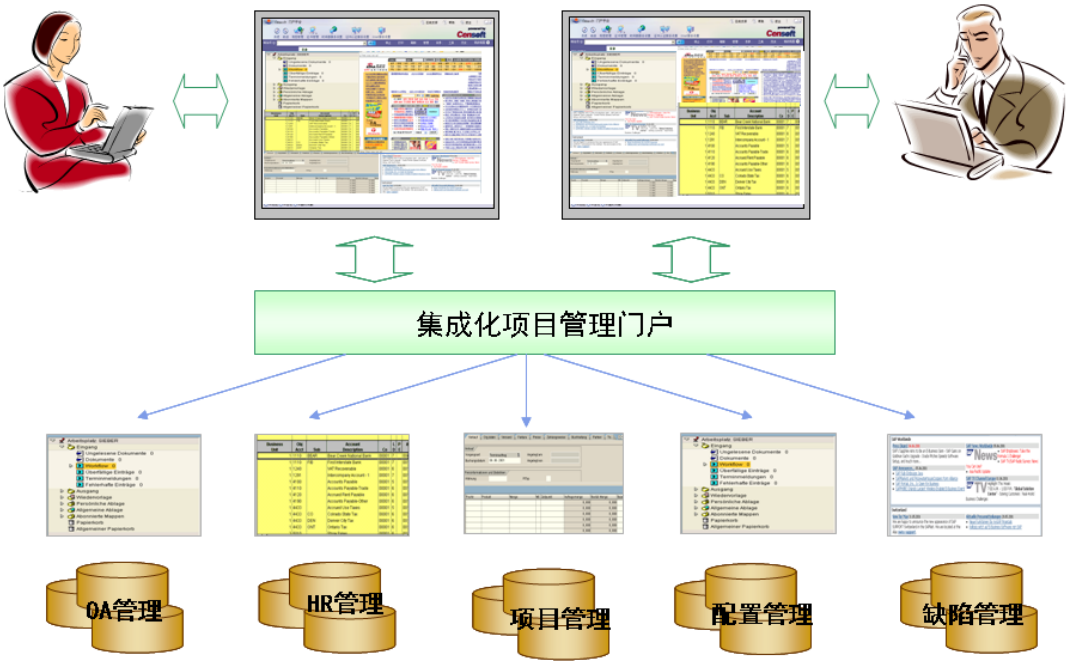


图 9-36　PM Suite 支持集成之 Web UI 方式

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 后端 多个自研客户端 兼容 MS Project 客户端
	API	多个 API
	插件	Eclipse 插件、 Visual Studio 插件
技术选型	构架风格	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。

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）。

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？