Broadview
www.broadview.com.cn

Pearson

架构整洁之道

Clean Architecture

[美] Robert C. Martin 著
孙宇聪 译

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

架构整洁之道

Clean Architecture

[美] Robert C. Martin 著
孙宇聪 译

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

内 容 简 介

《架构整洁之道》是创造“Clean神话”的Bob大叔在架构领域的登峰之作，围绕“架构整洁”这一重要导向，系统地剖析其缘起、内涵及应用场景，涵盖软件研发完整过程及所有核心架构模式。本书分为6部分，第1部分纲领性地提出软件架构设计的终极目标，描述软件架构设计的重点与模式；第2~4部分从软件开发中三个基础编程范式的定义和特征出发，进一步描述函数、组件、服务设计与实现的定律，以及它们是如何有效构建软件系统的整体架构的；第5部分从整洁架构的定义开始，详细阐述软件架构设计过程中涉及的方方面面，包括划分内部组件边界、应用常见设计模式、避开错误、降低成本、处理特殊情况等，并以实战案例将内容有机整合起来；第6部分讲述具体实现细节；附录则透过作者数十年的软件从业经历再次印证本书的观点。

对于每一位软件研发从业人员——无论从事的是具体编码实现、架构设计，还是软件研发管理，本书都是不可或缺的。

版权贸易合同登记号　图字：01-2017-7530

图书在版编目（CIP）数据

架构整洁之道 /（美）罗伯特·C.马丁（Robert C. Martin）著；孙宇聪译. —北京：电子工业出版社，2018.9
书名原文：Clean Architecture
ISBN 978-7-121-34796-2

Ⅰ. ①架… Ⅱ. ①罗… ②孙… Ⅲ. ①软件设计 Ⅳ. ①TP311.1

中国版本图书馆 CIP 数据核字(2018)第 168309 号

策划编辑：张春雨
责任编辑：付　睿
印　　刷：三河市良远印务有限公司
装　　订：三河市良远印务有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编：100036
开　　本：787×980　1/16　印张：22　字数：394 千字
版　　次：2018 年 9 月第 1 版
印　　次：2018 年 10 月第 2 次印刷
定　　价：99.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

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。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

推荐序一

在我心里，程序员可以分为三个层次：普通程序员、工程师和架构师。

普通程序员是编写代码的人。编写代码的方式有很多，只要能让程序跑起来，能正确地处理业务流程和对数据进行计算，就可以说“会编写代码”。程序员需要熟悉整个程序的逻辑及处理过程，需要熟悉程序语言的特性，还需要熟悉一些计算机操作系统的交互调用方式，才能写出从用户侧交互，到数据和业务逻辑处理，再到与计算机系统交互的代码，有效地把用户信息、数据、业务和计算机串联和拼装出来。

然而，其中一些程序员发现，只让代码跑起来是不够的，因为这个世界是不断变化的，他们发现自己需要花更多的时间来维护代码：增加新的需求，扩展原有的流程，修改已有的功能，优化性能……一个人完全维护不过来，还需要更多的人，于是代码还需要在不同人之间轮转；他们发现代码除了需要跑起来，还需要易读、易扩展、易维护，甚至可以直接重用。于是，这些人使用各种各样的手段和技术不断提高代码的易读性、可扩展性、可维护性和重用性。我们把这些有“洁癖”、有工匠精精、有修养的程序员叫作工程师，工程师不仅仅是在编写代码，他们会用工程的方法来编写代码，以便让编程开发更为高效和快速。他们把编程当成一种设计，一种工业设计，把代码模块化，让这些模块可以更容易地交互拼装和组织，让代码

排列整齐——阅读和维护这些代码就像看阅兵式一样舒心畅快。

但是故事还没完，这些拥有工匠精神的工程师们还是难以解决某些问题，这些人渐渐地发现，这个世界上有很多问题就像翘翘板一样，只能要一边，这一边上去了，另一边就下来了。就像要么用空间换时间，要么用时间换空间一样，你很难找到同时满足空间和时间要求的“双利解”；就像 CAP 的三选二的理论一样，这个世界不存在完美的解决方案，无论什么方案都有好的一面和不好的一面。而且，这些工程师还渐渐发现，每当引入一个新的技术来解决一个已有的问题时，这个新的技术就会带来更多的问题，问题就像有一个生命体一样，它们会不断地繁殖和进化。渐渐地，他们发现，问题的多少和系统的复杂度呈正比，而且不仅是线性正比，还可能呈级数正比，此时就越来越难做技术决定。但是有一些资深的工程师开始站出来挑战这些问题，有的基于业务分析给出平衡的方案，有的开始尝试设计更高级的技术，有的开始设计更灵活的系统，有的则开始简化和轻量化整个系统……这些高智商、经验足、不怕难的工程师们引领着整个行业前行。他们就是架构师！

感觉 Bob 大叔的系列著作好像也在走这个过程，《代码整洁之道》教你写出易读、可扩展、可维护、可重用的代码，《代码整洁之道：程序员的职业素养》教你怎样变成一个有修养的程序员，而《架构整洁之道》基本上是在描述软件设计的一些理论知识。《架构整洁之道》大体分成三个部分：编程范式（结构化编程、面向对象编程和函数式编程），设计原则（主要是 SOLID），以及软件架构（其中讲了很多高屋建瓴的内容）。总体来说，这本书中的内容可以让你从微观（代码层面）和宏观（架构层面）两个层面对整个软件设计有一个全面的了解。

但是，如果你想从这本书里找到一些可以立马解决具体问题的工程架构和技术，恐怕你会感到失望。这本书中更多的是一些基础的理论知识，看完后你可能会比较“无感”，因为这些基础知识对于生活在这个高速发展的喜欢快餐文化的社会中的人来说，可能很难理解其中的价值——大多数人的目标不是设计出一个优质的软件或架构，而是快速地解决一个具体的问题，完成自己的工作。然而，可能只有你碰过足够多的壁，掉过足够多的坑，经历过足够多的痛苦后，再来读这本书时，你才会发现本书中的这些“陈旧的知识”是多么充满智慧。而且，如果有一天，你像我这个老家伙一样，看到今天很多很多公司和年轻的程序员还在不断地掉坑和挣扎，你

就会明白这些知识的重要性了。

我个人觉得，这本书是架构方面的入门级读物，但也并不适合经验不足的人员学习，这本书更适合的读者群是，有 3~5 年编程经验、需要入门软件设计和架构的工程师或程序员。

最后，我想留下一个观点和一组问题。

观点：无论是微观世界的代码，还是宏观层面的架构，无论是三种编程范式还是微服务架构，它们都在解决一个问题——分离控制和逻辑。所谓控制就是对程序流转的与业务逻辑无关的代码或系统的控制（如多线程、异步、服务发现、部署、弹性伸缩等），所谓逻辑则是实实在在的业务逻辑，是解决用户问题的逻辑。控制和逻辑构成了整体的软件复杂度，有效地分离控制和逻辑会让你的系统得到最大的简化。

问题：如果你要成为一名架构师，你需要明确地区分几组词语（如何区分它们正是留给你的问题），否则你不可能成为一名合格的工程师或架构师。这几组词语是简单 vs.简陋、平衡 vs.妥协、迭代 vs.半成品。如果你不能很清楚地定义出其中的区别，那么你将很难做出正确的决定，也就不可有成为一名优秀的工程师或架构师。

我相信这个观点和这组问题将有助于你更好地阅读并理解这本书，也会让你进行更多的思考，带着思考读这本书，会让你学到更多！

陈皓

（@左耳朵耗子）

推荐序二

久远的教诲，古老的智慧

如果让你接手一套不稳定但要紧的在线系统，这套系统还有各种问题：变量命名非常随意，依赖逻辑错综复杂，层次结构乱七八糟，部署流程一塌糊涂，监控系统一片空白……你该怎么办？

前几年我就遇到了这种问题，我对着频发的故障仔细观察，发现了最关键的问题：如果放着不动，这套系统的核心功能还是相对稳定的，但经常会有一些外围需求要开发，这时由于原有的依赖逻辑和层次结构不够清楚，就会导致“牵一发而动全身”的情况，加上测试不完善，所以几乎每次外围功能上线更新，核心功能都会受影响，然后又要重复好几次“调试→改正→上线”的流程。

怎么办？大家说了很多办法：把单元测试都补全，重构代码拆分核心功能和非核心功能，跟业务方谈暂停需求……这些办法都很对，但是，都需要时间才能见效，而我们最缺的就是时间。

我提了一个很“笨”的办法：把所有“共享变量”都抽到 Redis 中进行读写，消灭本地副本，然后把稳定版本程序多部署几份，这样就可以多启动几个实例，将这些实例标记为 AB 两组。同时，在前面搭建代理服务，用于分流请求——核心功能请求分配到 A 组（程序基本不更新），外围功能请求分配到 B 组（程序按业务需

求更新）。这样做看起来有点多此一举——AB 两组都只有部分代码提供服务，而且要通过 Redis 共享状态，但是却实现了无论 B 组的程序如何更新，都不会影响 A 组所承载的核心服务的目的。

虽然当时不少人说“怎么能这样玩呢”，但它确实有效。当天部署，当天生效，在线服务迅速稳定下来，即便新开发的外围功能有问题，核心服务也不受任何影响。这样业务人员满意了，开发人员也可以安心对系统做改造了。

后来有不少人问我是怎么想到这个办法的，答案是：因为我是个老程序员，成长在面向对象的年代，运用 SOC（关注点分离）、SRP（单一职责原则）、OCP（开闭原则）这些东西对我来说就如同本能。具体到这个例子，无非就是识别关注点、隔离责任、保持核心关注点的封闭而已。

后来我才知道，我提出的这个方法有个专门的名字叫“蓝绿部署”。当然我自认是个老程序员，不懂这些新鲜概念也不太要紧。确实，如今不少程序员已经不认识 SOC、SRP、OCP、LSP 等“古老”的玩意了，大家熟悉的是各种语言、类库、框架、代码托管网站。互联网开发场景千变万化，技术一日千里，而面向对象在不少人的脑海里早就是弃之不用的老古董了。只有“老一辈”的程序员还记得那些古老的教诲，守着那些古拙的技巧。但是这些东西，总有一天会被时代淘汰吗？

实际上，这也是我初读《架构整洁之道》的疑惑。虽然 Bob 大叔这个名字对我们这些“老程序员”来说可谓如雷贯耳，之前针对一般性软件开发所著的《代码整洁之道》和《代码整洁之道：程序员的职业素养》也确实很受欢迎，但如今写架构，还从结构化编程、面向对象编程、函数式编程写起，还花时间解释 SRP、OCP、LSP 等原则，实在难掩“古老”的感觉。请问，它们和如今的“架构”有什么关系吗？

不过，如果你耐心读下去就会发现，还真有关系。按照 Bob 大叔的说法，所谓架构就是“用最小的人力成本来满足构建和维护系统需求”的设计行为。以前的面向对象系统和如今的分布式系统，在这一点上是完全一致的。听取久远的教诲，尊重古老的智慧，如今的架构师也会从中受益。不信？我们就拿经典的三个编程范式来举例，看看这些“老掉牙”的玩意儿和如今的架构设计有什么关联。

大家对结构化编程的一般理解是，由 if-else、switch-case 之类的语句组织程序

代码的编程方式，它杜绝了 goto 导致的混乱。但是从更深的层次上看，它也是一种设计范式，避免随意使用 goto，使用 if-else、switch-case 之类控制语句和函数、子函数组织起来的程序代码，可以保证程序的结构是清楚的，自顶向下层层细化，消灭了杂错，杜绝了混淆。

联系如今的分布式系统，我们在设计的时候，真的能够做到自顶向下层层细化吗？有多少次，我看到的系统设计图里，根本没有“层次”的概念，各个模块没有一致的层次划分，与子系统交互的不是子系统，而是一盘散沙式的接口，甚至接口之间随意互调、关系乱成一团麻的情况也时常出现，带来的就是维护和调试的噩梦。吹散历史的迷雾，不正是古老的 goto 陷阱的再现吗？

大家对面向对象编程的一般理解是，由封装、继承、多态三种特性支持的，包含类、接口等若干概念的编程方式。但是从更深的层次上看，它也是一种设计范式。多态大概算其中最神奇的特性了，程序员在确定接口时做好抽象，代码就可以很灵活，遇到新情况时，新写一个实现就可以无缝对接。

联系如今的分布式系统，我们在设计的时候，真的能够做到接口足够抽象、新模块能无缝对接吗？有多少次，我看到接口的设计非常随意，接口不是基于行为而是基于特定场景的实现，没有做适当的抽象，也没有为未来预留空间，直接导致契约僵硬死板。每新增一种终端呈现形式，整个内容生产流程就要大动干戈，这样的例子并不罕见。抹去历史的尘埃，这不正是“多态”出现之前的困境吗？

大家对函数式编程的一般理解是，以函数为基本单元，没有变量（更准确地说是不能重复赋值）也没有副作用的编程方式。但是从更深的层次上看，它彻底隔离了可变性，变量或者状态默认就是不可变的，如果要变化，则必须经过合理设计的专门机制来实现。所以，它也避免了死锁、状态冲突等众多麻烦。

联系如今的分布式系统，我们在设计的时候，真的能够彻底隔离可变性、避免状态冲突吗？有多少次，我看到状态或变量的修改接口大方暴露，被不经意（或者恶意）修改，导致奇怪的故障。Bob 大叔举了一个相当有趣的例子，如果又要保证操作原子性又要能精确还原各时刻的状态，有个办法是这样的：只提供 CR 操作，而不提供完整的 CRUD 操作（就像 MySQL 的 binlog 那样）。平时只要追加操作记录即可，各时刻的状态永远通过重放之前的操作记录得出，这样就彻底避免了状态

的错乱。这个办法看起来古怪，但我真的在之前的开发中用过（当然是在程序生命周期有限的场景下），而且真的从没出过错。

坦白说，看完《架构整洁之道》这本书，我心里好受点了。因为我发现，我们这些老程序员的知识其实没有过时，如今不少光鲜的架构其实要解决的还是那些古老的问题。多亏了 Bob 大叔的妙手点拨，我才能穿越时空，享受到“重新发现智慧”的愉悦。

当然，架构设计是一门复杂的学问，要综合考虑编码、质量、部署、发布、运维、排障、升级等等各种因素，做出权衡。好消息是，Bob 大叔的这本书覆盖面广，涉及各个方面，相信你认真读完全书一定会和我一样有不小的收获。唯一的问题是，你要适应这个老程序员的口吻和节奏：他当然也会拿如今流行的打车系统做例子，但他更熟悉的还是链接器、C 语言、UML 图等玩意。

不过我觉得，这都不是大问题。看得出类之间的依赖关系不合理，自然容易发现子系统之间的依赖关系不合理；搞得懂 UNIX 如何巧妙定义通用的 IO 设备，自然容易想到对 PC Web、Mobile Web、App 内的页面做适当抽象；认得清各线程、进程、链接库的职责，自然容易明白微服务也需要避免跨边界调用。更妙的是，从这种古老的视角看问题，往往更能摆脱细节的困扰，把握问题的核心。就像老子说的那样：治大国如烹小鲜。

噢，对了，“治大国如烹小鲜”也是久远的教诲，也包含着古老的智慧。

余　晟
公众号“余晟以为”（yurii-says）作者
现在沪江教育集团担任平台架构部负责人

序　言

软件架构（architecture）究竟是什么？

不论从哪个角度分析软件系统，都不可能面面俱到。如果从架构学角度来分析，在一定程度上能够做到抓大放小，把握住重点，但是也不可避免地会错失某些重要的细节信息。

软件架构学关注的的一个重点是组织结构（structure）。不管是讨论组件（Component）、类（Class）、函数（Function）、模块（Module），还是层级（Layer）、服务（Service）以及微观与宏观的软件开发过程，软件的组织结构都是我们的主要关注点。但是真实世界中的许多软件项目并不完全按照我们的信念和愿望生长——它们就像超大型国企那样，层层嵌套，缠绕成一团乱麻[1]。有的时候真的很难相信，软件项目的组织结构性也能像物理建筑那样一目了然，层次清晰。

物理建筑，不管其地基是石头还是水泥，形状是高大还是宽阔，风格是气势恢宏还是小巧玲珑，其组织结构都一目了然。物理建筑的组织结构必须遵守“受重力”这一自然规律，同时还要符合建筑材料自身的物理特性。软件项目则没有定律可以遵循。另外，物理建筑是用砖头、水泥、木头、钢铁或者玻璃等标准材料建成的，

1　原文是 the-big-ball-of-mud，见 https://en.wikipedia.org/wiki/Big_ ball_of_mud。——译者注

而大型软件项目往往是由小的软件组件构成的，这些软件组件又是由更小的软件组件构成的，层层堆叠，无穷无尽。

所以，当讨论软件架构时，要特别注意软件项目是具有递归（recursive）和分形（fractal）特点的，最终都要由一行行的代码组成。脱离了一行行的代码，脱离了具体的细节设计，架构设计就无从谈起。大型物理建筑通常可以用比例模型分层描述细节信息，但是软件项目内部结构是很难用模型分层描述的。软件项目也具有内部结构，但是其结构无论从数量上还是多样性上来说，都远远超过了物理建筑的结构。可以不夸张地说，软件开发比修建物理建筑需要更长、更专注的设计过程，软件架构师应该比建筑架构师更懂架构！

比例模型是深入人心的展示方式，但是不管某个 PowerPoint 图表中的彩色方块多么好看，多么简单易懂，它也无法完全代表一个软件的架构。它只能是该软件的架构的一个视图，而非全部。软件的架构并没有固定的展现形式，你所看到的每一个视图的背后都是架构师所做的层层抉择。一个视图包含了哪些部分，排除了哪些部分；用特殊形状和颜色强调了哪些部分，又有哪些部分被泛泛地一笔带过，甚至直接忽略，这些都是这个视图本身的特性。然而，每个视图都是对的，它们往往并没有优劣之分。[1]

虽然软件无法很好地用比例模型展示，但它还是要在现实世界中运行的。在设计软件架构的过程中，我们必须理解和遵守现实的约束条件。CPU 速度和网络带宽往往在很大程度上决定了系统的性能，而内存和存储空间的大小也会大幅影响代码的设计野心。

> 女士，这就是爱情的穷凶极恶之处，人的意愿是无穷的，而实际行动却处处受限。人的欲望是无止境的，行为却不得不遵从现实的限制。
>
> ——威廉·莎士比亚[2]

人类的整个经济活动都是存在于现实世界中的，所以我们可以利用现实世界的

1 这一段的意思是，软件的架构设计是多角度综合考虑的过程。对于某个软件的架构来说，不存在一个固定不变的视图，也不存在所谓的最佳试图。——译者注

2 出自《脱爱勒斯与克来西达》第三幕。——译者注

一些准则来衡量和推理软件开发过程中那些不好量化和物化的因素。

> 软件架构是系统设计过程中的重要设计决定的集合，可以通过变更成本来衡量每个设计决定的重要程度。
>
> ——Grady Booch

需要付出的时间、金钱和人力成本是区分软件架构规模大小的衡量标准，也可以用来区分架构设计和细节设计。同时，我们还可以依据这个信息来判断某个特定架构设计是好还是坏：一个好的架构，不仅要在某一特定时刻满足软件用户、开发者和所有者的需求，更要在一段时间内持续满足他们的后续需求。

> 如果你觉得好架构的成本太高，那你可以试试选择差的架构加上返工重来的成本。
>
> ——Brian Foote 和 Joseph Yoder

一个系统的常规变更不应该是成本高昂的，也不应该需要难以决策的大型设计调整，更不应该需要单独立项来推进。这些常规变更应该可以融入每日或者每周的日常系统维护中去完成。

我们怎么能够预知某个系统未来的变更需求，以便提前做准备呢？我们怎么能在没有水晶球与时光穿梭机的情况下，未卜先知，降低未来的变更成本呢？

> 所谓软件架构，就是你希望在项目一开始就能做对，但是却不一定能够做得对的决策的集合。
>
> ——Ralph Johnson

了解历史已经够难了，我们对现实的认知也不够可靠，预言未来就更难了。

这就是不同的软件开发理论的主要分歧点。

其中一条比较悲观阴暗的路线认为，只有权威和刚性才能带来强壮与稳定。如果某项变更成本高昂，那么就应该忽视它——变更背后的需求要么应该被抑制，要么就应该被丢到官僚主义的大机器中去绞碎。架构师的决定永远是完整的、彻底的，软件架构就是全体开发人员的敌托邦噩梦（Dystopia），永远是所有人沮丧的源泉。

另外一条路线则到处充斥着大量的投机性的通用设计。在这样的软件项目中到处都是硬编码的猜测性代码，到处是无穷无尽的参数，存在着成篇累牍的无效代码。维护这样的项目，肯定会遇到意外情况，而且无论预留多少资源都不够应付。

而本书试图探索的则是一条整洁路线。这条路线拥抱软件的灵活多变性，将其作为系统的一级设计目标。同时，我们也承认人类并不能全知全晓，但在信息不全的情况下人类仍然能够做出优良的决策。这条路线可以让我们多发挥优势，避开弱势。通过实际创造和探索，不停地提出问题和进行实验。优良的软件架构不是一成不变的，只有经过不断打磨和改进才能最终成就。

> 软件架构是一个猜想，只有通过实际实现和测量才能证实。
>
> ——Tom Gilb

遵循这条路线，我们需要用心，全神贯注，不停观察和思考，在原则指导下不断实践。虽然这可能听起来很麻烦、很慢，但是只要坚持走下去一定能够成功。

> 走快的唯一方法是先走好。
>
> ——Robert C. Martin

一起享受这个过程吧！

Kevlin Henney

2017 年 5 月

前　言

本书的名字叫作《架构整洁之道》，使用这个名字可谓是十分胆大，甚至可以说有点目中无人了。那么，为什么我会选择写这本书，并且使用这个名字呢？

自 1964 年，12 岁的我写下了人生的第一行代码算起，到 2016 年，我已经编程超过 50 年。在这段时间里，我自认为学到了构建软件系统的一些方法——并且我相信这些方法和经验对其他人应该有些价值。

我学习的途径是实际构建一些大大小小的软件系统。我写过小型的嵌入式系统，也构造过大型的批处理系统；我构建过实时控制系统，也构建过 Web 网页系统；我写过命令行程序、图形界面程序、进程管理程序、游戏、计费系统、通信系统、设计工具、画图工具等。

我写过单线程程序，也写过多线程程序；我写过由几个重型进程组成的应用，也写过由大量轻型进程组成的应用；我写过跨多个处理器的应用，还有数据库类、数值计算类和几何计算类应用，以及很多很多其他类型的应用。

回首过去，经历了这么多应用和系统的构建过程，我最意外的领悟是：

软件架构的规则是相同的！

我所构建的这些系统是千差万别的，为什么所有这些差异巨大的系统都遵守同样的软件架构规则呢？这里我得出的结论是，*软件架构规则和其他变量完全无关*。

回顾过去这半个世纪以来硬件系统产生的巨大变革，这个结论就更惊人了。我的编程生涯起步于像家用冰箱那么大的巨型机时代，它的 CPU 频率只有 0.5MHz，拥有 4KB 核心内存，32KB 磁盘存储，以及每秒只能传输 10 个字符的电传打字机接口。而现在，我正在一辆游览南非的观光车上敲这篇前言。我正在用一个拥有 4 核 i7 的 MacBook，每核 2.8GHz。这台笔记本电脑有 16GB 内存，1TB SSD 硬盘，可以用 2880×1800 虹膜显示屏展现高清视频。二者计算能力上的差距真的是天壤之别。粗略分析可知，这台 MacBook 至少比我半个世纪以前用的计算机强大 10^{22} 倍。

22 个数量级的差距是非常非常巨大的，从地球到半人马星系也只有 10^{22} 埃（angstrom，长度单位，主要用来描述原子尺寸与波长），你口袋里的零钱加起来所包含的电子数量也差不多为 10^{22} 个。而这个数字（注意还是*至少*）是我在一生中，所亲身经历的计算能力的提升。

计算能力发生了这么巨大的变化，但对我所写软件的影响有多大呢？软件尺寸当然变大了。我过去认为 2000 行的程序就很庞大了。毕竟这样的程序变成打孔卡片能装满一盒子，重量超过 10 磅。而现在，一个 10 万行的程序都不能算大程序了。

同时，软件性能当然也有大幅提升。我们现在可以轻轻松松地完成那些 1960 年只能幻想的事情。电影 *The Forbin Project*、*The Moon is a Harsh Mistress* 以及 *2001: A Space Odyssey* 都试图预言我们的现状，但是都没有成功。在这些电影中普遍展现的是获得了智能的巨型机器，而我们目前所拥有的计算机，虽然体积之小是当初难以想象的，却还仅仅只是机器。

同时，还有一点很重要，*今天的软件与过去的软件本质上仍然是一样的*。都是由 `if` 语句、赋值语句以及 `while` 循环组成的。

哦，你可能会抗议说我们现在有更好的编程语以及更先进的编程范式了。毕竟，我们现在都是用 Java、C#、Ruby 语言编写程序，并且大量采用面向对象编程方式。这没错，但是最终产生的代码仍然只是顺序结构、分支结构、循环结构的组合，这方面和 20 世纪 60 年代甚至 50 年代的程序是一模一样的。

如果深入研究计算机编程的本质，我们就会发现这 50 年来，计算机编程基本没有什么大的变化。编程语言稍微进步了一点，工具的质量大大提升了，但是计算机程序的基本构造没有什么变化。

如果一个 1966 年的计算机程序员时空穿梭来到 2016 年，在我的 MacBook 上用 IntelliJ 写 Java 程序，她[1]可能也就需要 24 小时来适应一下，然后很快就能照常工作了。Java 其实和 C 区别并不大，和 FORTRAN 也没那么大区别。

同样，如果我把你——读者传送回 1966 年，告诉你如何在一个每秒处理 10 个字符的终端上通过打孔纸带来编辑 PDP-8 代码，估计你最多也只需要 24 小时的适应时间。毕竟编程还是编程，代码并没有本质的变化。

这就是秘密所在：计算机代码没有变化，软件架构的规则也就一直保持了一致。软件架构的规则其实就是排列组合代码块的规则。由于这些代码块本质上没有变化，因此排列组合它们的规则也就不会变化。

年轻的一代程序员可能认为这些都是胡说。他们可能坚持认为现在所有东西都是崭新的、从来没有过的，过去的规则已经过时，不再适用了。这是一个非常大的错误。这些规则一直都没有变。虽然我们有了新的编程语言、新的编程框架、新的编程范式，但是软件架构的规则仍然和 1946 年阿兰·图灵写下第一行机器代码的时候一样。

当然，不一样的是，那时候我们还不知道规则是什么。所以我们一次一次地颠覆了它，并且为此一次一次地付出了代价。半个世纪过去了，我们终于可以说，现在我们对这些规则有一定程度的了解了。

写这本书就是为了讲述这些规则，这些永恒的、不变的软件架构规则。

1 这里用“她”，是因为当时计算机程序员大部分都是女性。

致　谢

下列这些人在本书编纂过程中提供了帮助，在此深表感谢，以下无特定顺序。

Chris Guzikowski
Chris Zahn
Matt Heuser
Jeff Overbey
Micah Martin
Justin Martin
Carl Hickman
James Grenning
Simon Brown
Kevlin Henney
Jason Gorman
Doug Bradbury
Colin Jones
Grady Booch
Kent Beck
Martin Fowler

Alistair Cockburn
James O. Coplien
Tim Conrad
Richard Lloyd
Ken Finder
Kris Iyer (CK)
Mike Carew
Jerry Fitzpatrick
Jim Newkirk
Ed Thelen
Joe Mabel
Bill Degnan

还有许多许多其他的人，这里不再一一列出。

在本书最后审校的过程中，当我读第 21 章的时候，回想起 Jim Weirich 明亮的大眼睛和富有旋律的笑声。Jim，祝你一切顺利!

读 者 服 务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- **提交勘误**：您对书中内容的修改意见可在 提交勘误 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动**：在页面下方 读者评论 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：http://www.broadview.com.cn/34796

关于作者

Robert C. Martin（Bob 大叔）从 1970 年起编程至今。他是 cleancoders.com 的联合创始人，该网站为软件开发者提供在线视频教育服务。同时，他还是 Uncle Bob Consulting LLC 的创始人，该公司为全球大型公司提供软件开发咨询、培训以及技能培训服务。同时，他现任 8th Light, Inc.的“首席匠人”一职，该公司是位于芝加哥的一家软件开发咨询公司。作者在各种行业周刊上发表了十余篇文章，同时也是国际会议和行业峰会经常邀请的演讲者。他曾任 *C++ Report* 的主编三年，并且曾任敏捷联盟（Agile Aliance）首任主席。

Martin 曾经编写和参与编辑了多本图书，包括 *The Clean Coder*、*Clean Code*、*UML for Java Programmers*、*Agile Software Development*、*Extreme Programming in Practice*、*More C++ Gems*、*Pattern Languages of Program Design 3*，以及 *Designing Object Oriented C++ Applications Using the Booch Method*。

关于技术审校者

鄢倩，ThoughtWorks 高级技术顾问，《Java 线程与并发编程实践》和《七周七语言 卷二》译者。是一名活跃在技术一线的技术顾问，为多家通信、金融企业提供基于敏捷精益原则的转型服务，在大型云服务系统中指导和实施过 DDD、持续集成和持续交付等实践。函数式编程爱好者。

目　录

第 1 部分　概述

第 2 部分　从基础构件开始：编程范式

第 3 部分 设计原则

第 4 部分 组件构建原则

第 5 部分　软件架构

第 6 部分　实现细节

第 1 部分

概述

编写并调试一段代码直到成功运行并不需要特别高深的知识和技能，现在的一名普通高中生都可以做到。有的大学生甚至通过拼凑一些 PHP 或 Ruby 代码就可以创办一个市值 10 亿美元的公司。想象一下，世界上有成群的初级程序员挤在大公司的隔板间里，日复一日地用蛮力将记录在大型问题跟踪系统里的巨型需求文档一点点转化为能实际运行的代码。他们写出的代码可能不够优美，但是确实能够正常工作。因为创造一个能正常运行的系统——哪怕只成功运行一次——还真不是一件特别困难的事。

但是将软件架构设计做好就完全另当别论了。软件架构设计是一件非常困难的事情，这通常需要大多数程序员所不具备的经验和技能。同时，也不是所有人都愿意花时间来学习和钻研这个方向。做一个好的软件架构师所需要的自律和专注程度可能会让大部分程序员始料未及，更别提软件架构师这个职业本身的社会认同感与人们投身其中的热情了。

但是，一旦将软件架构做好了，你就会立即体会到其中的奥妙：维持系统正常运转再也不需要成群的程序员了；每个变更的实施也不再需要巨大的需求文档和复杂

的任务追踪系统了；程序员们再也不用缩在全球各地的隔板间里，24×7（即每天 24 小时，每星期 7 天）地疯狂加班了。

采用好的软件架构可以大大节省软件项目构建与维护的人力成本。让每次变更都短小简单，易于实施，并且避免缺陷，用最小的成本，最大程度地满足功能性和灵活性的要求。

是的，这可能有点像童话故事一样不可信，但是这些又确实是我的亲身经历。我曾经见过因为采用了好的软件架构设计，使得整个系统构建更简单、维护更容易的情况。我也见过因为采用了好的软件架构设计，整个项目最终比预计所使用的人力资源更少，而且更快地完成了。我真真切切地体会过，好的软件架构设计为整个系统所带来的翻天覆地的变化，绝不忽悠。

请读者回头想想自己的亲身经历，你肯定经历过这样的情境：某个系统因为其组件错综复杂，相互耦合紧密，而导致不管多么小的改动都需要数周的恶战才能完成。又或是某个系统中到处充满了腐朽的设计和连篇累牍的恶心代码，处处都是障碍。再或者，你有没有见过哪个系统的设计如此之差，让整个团队的士气低落，用户天天痛苦，项目经理们手足无措？你有没有见过某个软件系统因其架构腐朽不堪，而导致团队流失，部门解散，甚至公司倒闭？作为一名程序员，你在编程时体会过那种生不如死的感觉吗？

以上这些我也都切身体会过。我相信绝大部分读者也或多或少会有共鸣。好的软件架构太难得了，我们职业生涯的大部分时间可能都在和差的架构做斗争，而没有机会一睹优美的架构究竟是什么样子。

第 1 章

设计与架构究竟是什么

一直以来，设计（Design）与架构（Architecture）这两个概念让大多数人十分迷惑——什么是设计？什么是架构？二者究竟有什么区别？

本书的一个重要目标就是要清晰、明确地对二者进行定义。首先我要明确地说，二者没有任何区别。一丁点区别都没有！

“架构”这个词往往使用于“高层级”的讨论中。这类讨论一般都把“底层”的实现细节排除在外。而“设计”一词，往往用来指代具体的系统底层组织结构和实现的细节。但是，从一个真正的系统架构师的日常工作来看，这样的区分是根本不成立的。

以给我设计新房子的建筑设计师要做的事情为例。新房子当然是存在着既定架构的，但这个架构具体包含哪些内容呢？首先，它应该包括房屋的形状、外观设计、垂直高度、房间的布局，等等。但是，如果查看建筑设计师使用的图纸，会发现其中也充斥着大量的设计细节。譬如，我们可以看到每个插座、开关以及每个电灯具体的安装位置，同时也可以看到某个开关与所控制的电灯的具体连接信息；我们也能看到壁炉的具体安装位置，热水器的大小和位置信息，甚至是污水泵的位置；同时也可以看到关于墙体、屋顶和地基都有非常详细的建造说明。

总的来说，架构图里实际上包含了所有的底层设计细节，这些细节信息共同支撑了顶层的架构设计，底层设计信息和顶层架构设计共同组成了整个房屋的架构文档。

软件设计也是如此。底层设计细节和高层架构信息是不可分割的。它们组合在一起，共同定义了整个软件系统，缺一不可。所谓的底层和高层本身就是一系列决策组成的连续体，并没有清晰的分界线。

目标是什么

所有这些决策的终极目标是什么呢？一个好的软件设计的终极目标是什么呢？

就像我之前描述过的：

> 软件架构的终极目标是，用最小的人力成本来满足构建和维护该系统的需求。

一个软件架构的优劣，可以用它满足用户需求所需要的成本来衡量。如果该成本很低，并且在系统的整个生命周期内一直都能维持这样的低成本，那么这个系统的设计就是优良的。如果该系统的每次发布都会提升下一次变更的成本，那么这个设计就是不好的。就这么简单。

案例分析

下面来看一个真实案例，该案例中的数据均来源于一个要求匿名的真实公司。

首先，我们来看一下工程师团队规模的增长。你肯定认为这个增长趋势是特别可喜的，像图 1.1 中的这种增长线条一定是公司业务取得巨大成功的直观体现。

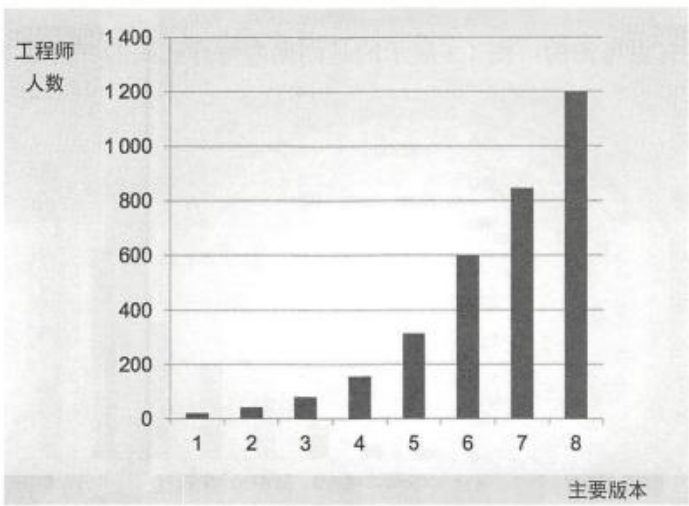


图 1.1：工程师团队的增长趋势

摘自 Jason Gorman 的 PPT，已授权。

现在再让我们来看一下整个公司同期的生产效率（productivity），这里用简单的代码行数作为指标（参见图 1.2）。

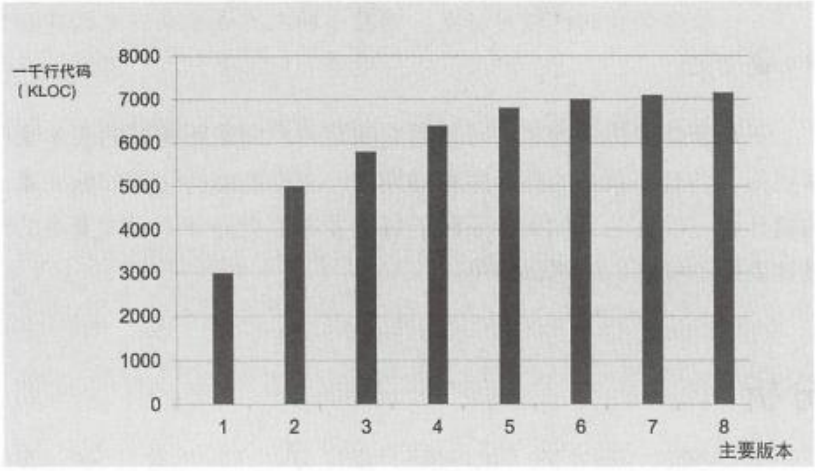


图 1.2：同期生产效率

这明显是有问题的。伴随着产品的每次发布，公司的工程师团队在持续不断地扩展壮大，但是仅从代码行数的增长来看，该产品却正在逐渐陷入困境。

还有更可怕的：图 1.3 展示的是同期内每行代码的变更成本。

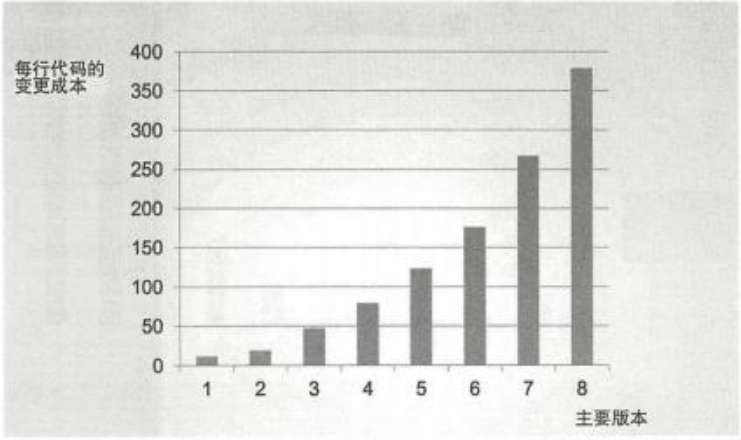


图 1.3：同期平均每行代码的变更成本

显然，图中展示的趋势是不可持续的。不管公司现在的利润率有多高，图中线条表明，按这个趋势下去，公司的利润会被一点点榨干，整个公司会因此陷入困境，甚至直接关门倒闭。

究竟是什么因素造成了生产力的大幅变化呢？为什么第 8 代产品的构建成本要比第 1 代产品高 40 倍？

乱麻系统的特点

我们在这里看到的是一个典型的乱麻系统。这种系统一般都是没有经过设计，匆匆忙忙被构建起来的。然后为了加快发布的速度，拼命地往团队里加入新人，同时加上决策层对代码质量提升和设计结构优化存在着持续的、长久的忽视，这种状态能持续下去就怪了。

图 1.4 展示了系统开发者的切身体会。他们一开始的效率都接近 100%，然而伴随着每次产品的发布，他们的生产力直线下降。到了产品的第 4 版本时，很明显大家的生产力已经不可避免地趋近为零了。

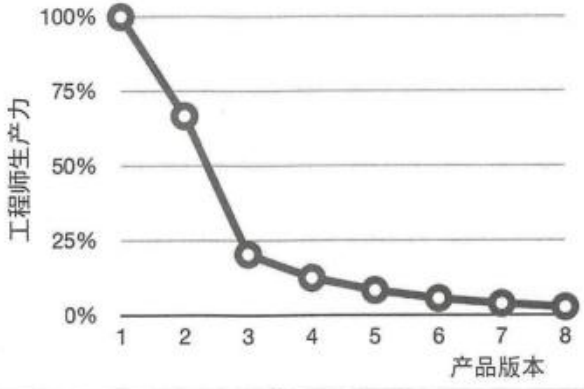


图 1.4：工程师生产力/产品版本的对比图

对系统的开发者来说，这会带来很大的挫败感，因为团队中并没有人偷懒，每个人还都是和之前一样在拼命工作。

然而，不管他们投入了多少个人时间，救了多少次火，加了多少次班，他们的产出始终上不去。工程师的大部分时间都消耗在对现有系统的修修补补上，而不是真正完成实际的新功能。这些工程师真正的任务是：拆了东墙补西墙，周而往复，偶尔有精力能顺便实现一点小功能。

管理层视角

如果你觉得开发者们这样就已经够苦了，那么就再想想公司高管们的感受吧！请看图 1.5，该部门月工资同期图。

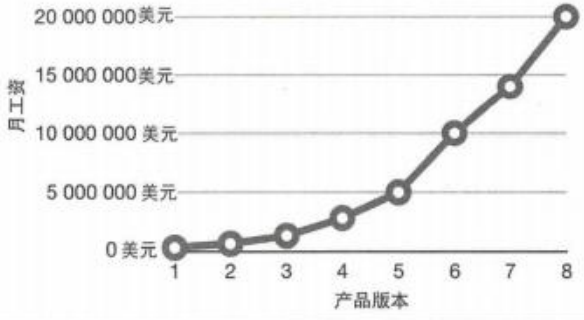


图 1.5：平均月工资/产品版本的对比图

如你所见，产品的第 1 版是在月总工资 10 万美元左右的时候上线的。第 2 版又花掉了几十万美元。当发布第 8 版的时候，部门月工资已经达到了 2 千万美元，而且还在持续上升。

也许我们可以指望该公司的营收增长远远超出成本增长，这样公司就还能维持正常运转。但是这么惊人的曲线还是值得我们深入挖掘其中存在的巨大问题的。

现在，只要将图 1.5 的月工资曲线和图 1.2 的每次发布代码行数曲线对比一下，任何一个理性的 CEO 都会一眼看出其中的问题：最开始的十几万美元工资给公司带来了很多新功能、新收益，而最后的 2 千万美元几乎全打了水漂。应立刻采取行动解决这个问题，刻不容缓。

但是具体采取什么样的行动才能解决问题呢？究竟问题出在哪里？是什么造成了工程师生产力的直线下降？高管们除了跺脚、发飙，还能做什么呢？

问题到底在哪里

大约 2600 年前，《伊索寓言》里写到了龟兔赛跑的故事。这个故事的主题思想可以归纳为以下几种：

1．慢但是稳，是成功的秘诀。

2．该比赛并不是拼谁开始跑得快，也不是拼谁更有力气的。

3．心态越急，反而跑得越慢。

这个故事本身揭露的是过度自信的愚蠢行为。兔子由于对自己速度的过度自信，没有把乌龟当回事，结果乌龟爬过终点线取得胜利的时候，它还在睡觉。

这和现代软件研发工作有点类似，现在的软件研发工程师都有点过于自信。哦，当然，他们确实不会偷懒，一点也不。但是他们真正偷懒的地方在于——持续低估那些好的、良好设计的、整洁的代码的重要性。

这些工程师们普遍用一句话来欺骗自己：“我们可以未来再重构代码，产品上线最重要！”但是结果大家都知道，产品上线以后重构工作就再没人提起了。市场的压力永远也不会消退，作为首先上市的产品，后面有无数的竞争对手追赶，必须要比他们跑得更快才能保持领先。

所以，重构的时机永远不会再有了。工程师们忙于完成新功能，新功能做不完，哪有时间重构老的代码？循环往复，系统成了一团乱麻，生产效率持续直线下降，直至为零。

结果就像龟兔赛跑中过于自信的兔子一样，软件研发工程师们对自己保持高产出的能力过于自信了。但是乱成一团的系统代码可没有休息时间，也不会放松。如果不严加提防，在几个月之内，整个研发团队就会陷入困境。

工程师们经常相信的另外一个错误观点是：“在工程中容忍糟糕的代码存在可以在短期内加快该工程上线的速度，未来这些代码会造成一些额外的工作量，但是并没有什么大不了。”相信这些鬼话的工程师对自己清理乱麻代码的能力过于自信了。但是更重要的是，他们还忽视了一个自然规律：无论是从短期还是长期来看，胡乱编写代码的工作速度其实比循规蹈矩更慢。

图 1.6 展示的是 Jason Gorman 进行的一次为期 6 天的实验。在该实验中，Jaosn 每天都编写一段代码，功能是将一个整数转化为相应罗马数字的字符串。当事先定义好的一个测试集完全通过时，即认为当天工作完成。每天实验的时长不超过 30 分

钟。第一天、第三天和第五天，Jason 在编写代码的过程中采用了业界知名的优质代码方法论：测试驱动开发（TDD），而其他三天他则直接从头开始编写代码。

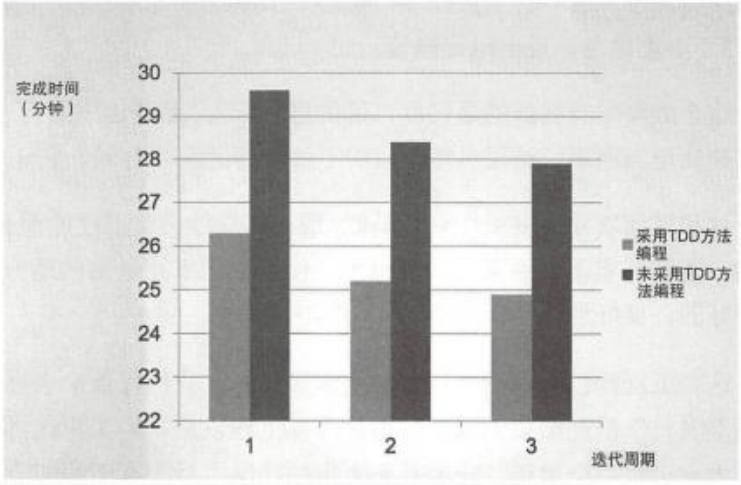


图 1.6：采用 TDD 方法与未采用 TDD 方面编程在每个迭代周期上的耗时对比图

首先，我们要关注的是图 1.6 中那些柱状图。很显然，日子越往后，完成工作所需的时间就越少。同时，我们也可以看到当人们采用了 TDD 方法编程后，一般就会比未采用 TDD 方法编程少用 10%的时间，并且采用 TDD 方法编程时最差的一天也比未采用 TDD 方法编程时最好的一天用时要短。

对于这个结果，有些人可能会觉得挺意外的。但是对常年关注软件开发本质的人来说，它其实揭示了软件开发的一个核心特点：

要想跑得快，先要跑得稳。

综上所述，管理层扭转局面的唯一选择就是扭转开发者的观念，让他们从过度自信的兔子模式转变回来，为自己构建的乱麻系统负起责任来。

当然，某些软件研发工程师可能会认为挽救一个系统的唯一办法是抛弃现有系统，设计一个全新的系统来替代。但是这里仍然没有逃离过度自信。试问：如果是工程师的过度自信导致了目前的一团乱麻，那么，我们有什么理由认为让他们从头开始，结果就会更好呢？

过度自信只会使得重构设计陷入和原项目一样的困局中。

本章小结

不管怎么看，研发团队最好的选择是清晰地认识并避开工程师们过度自信的特点，开始认真地对待自己的代码架构，对其质量负责。

要想提高自己软件架构的质量，就需要先知道什么是优秀的软件架构。而为了在系统构建过程中采用好的设计和架构以便减少构建成本，提高生产力，又需要先了解系统架构的各种属性与成本和生产力的关系。

这就是这本书的主题。本书为读者描述了什么是优秀的、整洁的软件架构与设计，读者可以参考这些设计来构建一个长期稳定的、持久优秀的系统。

第 2 章

两个价值维度

对于每个软件系统，我们都可以通过行为和架构两个维度来体现它的实际价值。软件研发人员应该确保自己的系统在这两个维度上的实际价值都能长时间维持在很高的状态。不幸的是，他们往往只关注一个维度，而忽视了另外一个维度。更不幸的是，他们常常关注的还是错误的维度，这导致了系统的价值最终趋降为零。

行为价值

软件系统的行为是其最直观的价值维度。程序员的工作就是让机器按照某种指定方式运转，给系统的使用者创造或者提高利润。程序员们为了达到这个目的，往往需要帮助系统使用者编写一个对系统功能的定义，也就是需求文档。然后，程序员们再把需求文档转化为实际的代码。

当机器出现异常行为时，程序员要负责调试，解决这些问题。

大部分程序员认为这就是他们的全部工作。他们的工作是且仅是：按照需求文档编写代码，并且修复任何 Bug。这真是大错特错。

架构价值

软件系统的第二个价值维度，就体现在软件这个英文单词上：software。“ware”的意思是“产品”，而“soft”的意思，不言而喻，是指软件的灵活性。

软件系统必须保持灵活。软件发明的目的，就是让我们可以以一种灵活的方式来改变机器的工作行为。对机器上那些很难改变的工作行为，我们通常称之为硬件（*hard*ware）。

为了达到软件的本来目的，软件系统必须够“软”——也就是说，软件应该容易被修改。当需求方改变需求的时候，随之所需的软件变更必须可以简单而方便地实现。变更实施的难度应该和变更的范畴（scope）成等比关系，而与变更的具体形状（*shape*）无关。

需求变更的范畴与形状，是决定对应软件变更实施成本高低的关键。这就是为什么有的代码变更的成本与其实现的功能改变不成比例。这也是为什么第二年的研发成本比第一年的高很多，第三年又比第二年更高。

从系统相关方（Stakeholder）的角度来看，他们所提出的一系列的变更需求的范畴都是类似的，因此成本也应该是固定的。但是从研发者角度来看，系统用户持续不断的变更需求就像是要求他们不停地用一堆不同形状的拼图块，拼成一个新的形状。整个拼图的过程越来越困难，因为现有系统的形状永远和需求的形状不一致。

我们在这里使用了“形状”这个词，这可能不是该词的标准用法，但是其寓意应该很明确。毕竟，软件工程师们经常会觉得自己的工作就是把方螺丝拧到圆螺丝孔里面。

问题的实际根源当然就是系统的架构设计。如果系统的架构设计偏向某种特定的“形状”，那么新的变更就会越来越难以实施。所以，好的系统架构设计应该尽可能做到与“形状”无关。

哪个价值维度更重要

那么，究竟是系统行为更重要，还是系统架构的灵活性更重要？哪个价值更大？系统正常工作更重要，还是系统易于修改更重要？

如果这个问题由业务部门来回答，他们通常认为系统正常工作很重要。系统开发人员常常也就跟随采取了这种态度。但是这种态度是错误的。下面我就用简单的逻辑推导来证明这个态度的错误性。

- 如果某程序可以正常工作，但是无法修改，那么当需求变更的时候它就不再能够正常工作了，我们也无法通过修改让它能继续正常工作。因此，这个程序的价值将成为 0。
- 如果某程序目前无法正常工作，但是我们可以很容易地修改它，那么将它改好，并且随着需求变化不停地修改它，都应该是很容易的事。因此，这个程序会持续产生价值。

当然，上面的逻辑论断可能不足以说服大家，毕竟理论上没有什么程序是不能修改的。但是，现实中有一些系统确实无法更改，因为其变更实施的成本会远远超过变更带来的价值。你在实际工作中一定遇到过很多这样的例子。

如果你问业务部门，是否想要能够变更需求，他们的回答一般是肯定的，而且他们会增加一句：完成现在的功能比实现未来的灵活度更重要。但讽刺的是，如果事后业务部门提出了一项需求，而你的预估工作量大大超出他们的预期，这帮家伙通常会对你放任系统混乱到无法变更的状态而勃然大怒。

艾森豪威尔矩阵

我们来看美国前总统艾森豪威尔的紧急/重要矩阵（参见图 2.1），面对这个矩阵，艾森豪威尔曾说道：

> 我有两种难题：紧急的和重要的，而紧急的难题永远是不重要的，重要的难题永远是不紧急的。[1]

图 2.1：艾森豪威尔矩阵

虽然老调重弹，但其中的道理依然成立。确实，紧急的事情常常没那么重要，而重要的事情则似乎永远也排不上优先级。

软件系统的第一个价值维度：系统行为，是紧急的，但是并不总是特别重要。

软件系统的第二个价值维度：系统架构，是重要的，但是并不总是特别紧急。

1　来自他于 1954 年在西北大学（Northwest University）的演讲。

当然，我们会有些重要且紧急的事情，也会有一些事情不重要也不紧急。最终我们应将这四类事情进行如下排序：

1. 重要且紧急
2. 重要不紧急
3. 不重要但紧急
4. 不重要且不紧急

在这里你可以看到，软件的系统架构——那些重要的事情——占据了该列表的前两位，而系统行为——那些紧急的事情——只占据了第一和第三位。

业务部门与研发人员经常犯的共同错误就是将第三优先级的事情提到第一优先级去做。换句话说，他们没有把真正紧急并且重要的功能和紧急但是不重要的功能分开。这个错误导致了重要的事被忽略了，重要的系统架构问题让位给了不重要的系统行为功能。

但研发人员还忘了一点，那就是业务部门原本就是没有能力评估系统架构的重要程度的，这本来就应该是研发人员自己的工作职责！所以，平衡系统架构的重要性与功能的紧急程度这件事，是软件研发人员自己的职责。

为好的软件架构而持续斗争

为了做好上述职责，软件团队必须做好斗争的准备——或者说“长期抗争”的准备。现状就是这样。研发团队必须从公司长远利益出发与其他部门抗争，这和管理团队的工作一样，甚至市场团队、销售团队、运营团队都是这样。公司内部的抗争本来就是无止境的。

有成效的软件研发团队会迎难而上，毫不掩饰地与所有其他的系统相关方进行平等的争吵。请记住，作为一名软件开发人员，你也是相关者之一。软件系统的可维护性需要由你来保护，这是你角色的一部分，也是你职责中不可缺少的一部分。公司雇你的很大一部分原因就是需要有人来做这件事。

如果你是软件架构师，那么这项工作就加倍重要了。软件架构师这一职责本身就应更关注系统的整体结构，而不是具体的功能和系统行为的实现。软件架构师必须创建出一个可以让功能实现起来更容易、修改起来更简单、扩展起来更轻松的软件架构。

请记住：如果忽视软件架构的价值，系统将会变得越来越难以维护，终会有一天，系统将会变得再也无法修改。如果系统变成了这个样子，那么说明软件开发团队没有和需求方做足够的抗争， 没有完成自己应尽的职责。

第 2 部分

从基础构件开始：编程范式

任何软件架构的实现都离不开具体的代码，所以我们对软件架构的讨论应该从第一行被写下的代码开始。

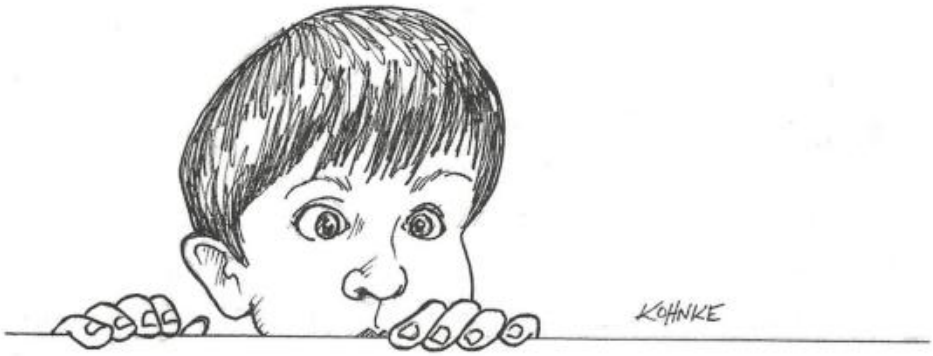
1938 年，阿兰·图灵为现代计算机编程打下了地基。尽管他并不是第一个发明可编程机器的人，但却是第一个提出“程序即数据”的人。到 1945 年时，图灵已经在真实计算机上编写真实的、我们现在也能看懂的计算机程序了。这些程序中用到了循环、分支、赋值、子调用、栈等如今我们都很熟悉的结构。而图灵用的编程语言就是简单的二进制数序列。

从那时到现在，编程领域历经了数次变革，其中我们都很熟悉的就是编程语言的变革。首先是在 20 世纪 40 年代末期出现了汇编器（assembler），它能自动将一段程序转化为相应的二进制数序列，大幅解放了程序员。然后是 1951 年，Grace Hopper 发明了 A0，这是世界上第一个编译器（*compiler*）。事实上，编译器这个名字就是他定义和推广使用的。再接着就到了 1953 年，那一年 FORTRAN 面世了（就在我出生的第二年）。接下来就是层出不穷的新编程语言了——COBOL、PL/1、SNOBOL、C、Pascal、C++、Java 等等，不胜枚举。

除此之外，计算机编程领域还经历了另外一个更巨大、更重要的变革，那就是*编程范式*（*paradigm*）的变迁。编程范式指的是程序的编写模式，与具体的编程语言关系相对较小。这些范式会告诉你应该在什么时候采用什么样的代码结构。直到今天，我们也一共只有三个编程范式，而且未来几乎不可能再出现新的，接下来我们就看一下为什么。

第 3 章

编程范式总览


KOHNKE


LIBERTY
IN GOD WE TRUST
2017
LIBERTY
IN GOD WE TRUST
2017

本章将讲述三个编程范式，它们分别是结构化编程（structured programming）、面向对象编程（object-oriented programming）以及函数式编程（functional programming）。

结构化编程

结构化编程是第一个普遍被采用的编程范式（但是却不是第一个被提出的），由 Edsger Wybe Dijkstra 于 1968 年最先提出。与此同时，Dijkstra 还论证了使用 `goto` 这样的无限制跳转语句将会损害程序的整体结构。接下来的章节我们还会说到，也是这位 Dijkstra 最先主张用我们现在熟知的 `if/then/else` 语句和 `do/while/until` 语句来代替跳转语句的。

我们可以将结构化编程范式归结为一句话：

> 结构化编程对程序控制权的直接转移进行了限制和规范。

面向对象编程

说到编程领域中第二个被广泛采用的编程范式，当然就是面向对象编程了。事实上，这个编程范式的提出比结构化编程还早了两年，是在 1966 年由 Ole Johan Dahl 和 Kriste Nygaard 在论文中总结归纳出来的。这两个程序员注意到在 ALGOL 语言中，函数调用堆栈（call stack frame）可以被挪到堆内存区域里，这样函数定义的本地变量就可以在函数返回之后继续存在。这个函数就成为了一个类（class）的构造函数，而它所定义的本地变量就是类的成员变量，构造函数定义的嵌套函数就成为了成员方法（method）。这样一来，我们就可以利用多态（polymorphism）来限制用户对函数指针的使用。

在这里，我们也可以用一句话来总结面向对象编程：

> 面向对象编程对程序控制权的间接转移进行了限制和规范。

函数式编程

尽管第三个编程范式是近些年才刚刚开始被采用的，但它其实是三个范式中最先被发明的。事实上，函数式编程概念是基于与阿兰·图灵同时代的数学家 Alonzo Church 在 1936 年发明的 λ 演算的直接衍生物。1958 年 John Mccarthy 利用其作为基础发明了 LISP 语言。众所周知，λ 演算法的一个核心思想是不可变性——某个符号所对应的值是永远不变的，所以从理论上来说，函数式编程语言中应该是没有赋值语句的。大部分函数式编程语言只允许在非常严格的限制条件下，才可以更改某个变量的值。

因此，我们在这里可以将函数式编程范式总结为下面这句话：

> 函数式编程对程序中的赋值进行了限制和规范。

仅供思考

如你所见，我在介绍三个编程范式的时候，有意采用了上面这种格式，目的是凸显每个编程范式的实际含义——它们都从某一方面限制和规范了程序员的能力。没有一个范式是增加新能力的。也就是说，每个编程范式的目的都是设置限制。这些范式主要是为了告诉我们不能做什么，而不是可以做什么。

另外，我们应该认识到，这三个编程范式分别限制了 `goto` 语句、函数指针和赋值语句的使用。那么除此之外，还有什么可以去除的吗？

没有了。因此这三个编程范式可能是仅有的三个了——如果单论去除能力的编程范式的话。支撑这一结论的另外一个证据是，三个编程范式都是在 1958 年到 1968 年这 10 年间被提出来的，后续再也没有新的编程范式出现过。

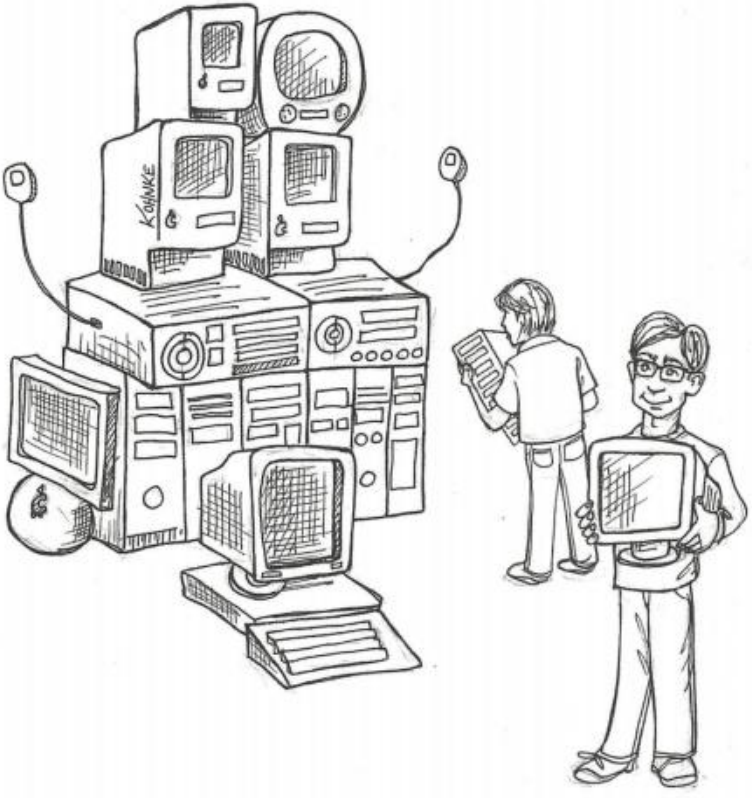
本章小结

大家可能会问，这些编程范式的历史知识与软件架构有关系吗？当然有，而且关系相当密切。譬如说，多态是我们跨越架构边界的手段，函数式编程是我们规范和限制数据存放位置与访问权限的手段，结构化编程则是各模块的算法实现基础。

这和软件架构的三大关注重点不谋而合：功能性、组件独立性以及数据管理。

第 4 章

结构化编程



Edsger Wybe Dijkstra 于 1930 年出生在荷兰鹿特丹。生于乱世，他亲身经历了第二次世界大战中的鹿特丹大轰炸、德国占领荷兰等事件。1948 年，他以数学、物理、化学以及生物全满分的成绩高中毕业。1952 年 3 月，年仅 21 岁的 Dijkstra（此时距离我出生还有 9 个月时间）入职荷兰阿姆斯特丹数学中心，成为了荷兰的第一个程序员。

1955 年，在从事编程工作 3 年之后，当时还是一个学生的 Dijkstra 就认为编程相比理论物理更有挑战性，因此他选择将编程作为终身职业。

1957 年，Dijkstra 与 Maria Debets 结婚了。在当时的荷兰，新郎新娘必须在结婚仪式上公布自己的职业。而当时的荷兰官方政府拒绝承认“程序员”这一职业，因为他们从来没有听说过。最终 Dijkstra 不得不继续使用“理论物理学家”这一职位名称。

Dijkstra 和他的老板 Adriaan van Wijingaarden 曾经讨论过将“程序员”当作终身职业这件事，Dijkstra 最担心的是由于没有人认真地对待过编程这件事或者将它当作是一门学术学科对待，他的科研成果可能将不会得到认真对待。而 Adriaan 则建议 Dijkstra：为什么不亲自去开创这门学科呢？

当时还是真空管阶段。计算机体积巨大，运行缓慢，还非常容易出故障，功能（与今天对比）十分有限。人们还是直接使用二进制数，或者使用非常原始的汇编语言编程。计算机的输入方式则还是用纸卷带或者是打孔卡片。要想执行完整的编辑、编译、测试流程是非常耗时的，通常需要数小时或者数天才能完成。

Dijkstra 就是在这样原始的条件下做出其非凡的成就的。

可推导性

Dijkstra 很早就得出的结论是：编程是一项难度很大的活动。一段程序无论复杂与否，都包含了很多的细节信息。如果没有工具的帮助，这些细节的信息是远远超过一个程序员的认知能力范围的。而在一段程序中，哪怕仅仅是一个小细节的错误，也会造成整个程序出错。

Dijkstra 提出的解决方案是采用数学推导方法。他的想法是借鉴数学中的公理（Postulate）、定理（Theorem）、推论（Corollary）和引理（Lemma），形成一种欧几里得结构。Dijkstra 认为程序员可以像数学家一样对自己的程序进行推理证明。换句话说，程序员可以用代码将一些已证明可用的结构串联起来，只要自行证明这些额外代码是正确的，就可以推导出整个程序的正确性。

当然，在这之前，必须先展示如何推导证明简单算法的正确性，这本身就是一件极具挑战性的工作。

Dijkstra 在研究过程中发现了一个问题：`goto` 语句的某些用法会导致某个模块无法被递归拆分成更小的、可证明的单元，这会导致无法采用分解法来将大型问题进一步拆分成更小的、可证明的部分。

`goto` 语句的其他用法虽然不会导致这种问题，但是 Dijkstra 意识到它们的实际效果其实和更简单的分支结构 `if-then-else` 以及循环结构 `do-while` 是一致的。如果代码中只采用了这两类控制结构，则一定可以将程序分解成更小的、可证明的单元。

事实上，Dijkstra 很早就知道将这些控制结构与顺序结构的程序组合起来很有用。因为在两年前，Bohm 和 Jocopini 刚刚证明了人们可以用顺序结构、分支结构、循环结构这三种结构构造出任何程序。

这个发现非常重要：因为它证明了我们构建可推导模块所需要的控制结构集与构建所有程序所需的控制结构集的最小集是等同的。这样一来，结构化编程就诞生了。

Dijkstra 展示了顺序结构的正确性可以通过枚举法证明，其过程与其他一般的数学推导过程是一样的：针对序列中的每个输入，跟踪其对应的输出值的变化就可以了。

同样的，Dijkstra 利用枚举法又证明了分支结构的可推导性。因为我们只要能用枚举法证明分支结构中每条路径的正确性，自然就可以推导出分支结构本身的正确性。

循环结构的证明过程则有些不同，为了证明一段循环程序的正确性，Dijkstra 需要采用*数学归纳法*。具体来说就是，首先要用枚举法证明循环 1 次的正确性。接下来再证明如果循环 *N* 次是正确的，那么循环 *N*+1 次也同样也是正确的。最后还要用枚举法证明循环结构的起始与结束条件的正确性。

尽管这些证明过程本身非常复杂和烦琐，但确实是完备的。有了这样的证明过程，用欧几里得层级构造定理的方式来验证程序正确性的目标，貌似近在咫尺了。

goto 是有害的

1968 年，Dijkstra 曾经给 CACM 的编辑写过一封信。这封信后来发表于 CACM 3 月刊，标题是 *Go To Statement Considered Harmful*[1]，Dijkstra 在信中具体描绘了他对三种控制结构的看法。

这可捅了个大篓子。由于当时还没有互联网，大家还不能直接上网发帖来对 Dijkstra 进行冷嘲热讽，他们唯一能做的，也是大部分人的选择，就是不停地给各种公开发表的报刊的编辑们写信。

可想而知，有的信件的措辞并不那么友善，甚至是非常负面的。但是，也不乏强烈支持者。总之，这场火热的争论持续了超过 10 年。

当然，这场辩论最终还是逐渐停止了。原因很简单：Dijkstra 是对的。随着编程语言的演进，`goto` 语句的重要性越来越小，最终甚至消失了。如今大部分的现代编程语言中都已经没有了 `goto` 语句。哦，对了，LISP 里从来就没有过！

现如今，无论是否自愿，我们都是结构化编程范式的践行者了，因为我们用的编程语言基本上都已经禁止了不受限制的直接控制转移语句。

或许有些人会指出，Java 中的带命名的 `break` 语句或者 Exception 都和 `goto` 很类似。事实上，这些语法结构与老的编程语言（类似 FORTRAN 和 COBOL）中的完全无限制的 `goto` 语句根本不一样。就算那些还支持 `goto` 关键词的编程语言

1 错误地使用 goto 语句是有害的。——译者注

也通常限制了 goto 的目标不能超出当前函数范围。

功能性降解拆分

既然结构化编程范式可将模块递归降解拆分为可推导的单元，这就意味着模块也可以按功能进行降解拆分。这样一来，我们就可以将一个大型问题拆分为一系列高级函数的组合，而这些高级函数各自又可以继续被拆分为一系列低级函数，如此无限递归。更重要的是，每个被拆分出来的函数也都可以用结构化编程范式来书写。

以此为理论基础，在 20 世纪 70 年代晚期到 80 年代中期出现的结构化分析与结构化设计工作才能广为人知。Ed Yourdon、Larry Constantine、Tom DeMarco 以及 Meilir Page Jones 在这期间为此做了很多推广工作。通过采用这些技巧，程序员可以将大型系统设计拆分成模块和组件，而这些模块和组件最终可以拆分为更小的、可证明的函数。

形式化证明没有发生

但是，人人都用完整的形式化证明的一天没有到来。大部分人不会真的按照欧几里得结构为每个小函数书写冗长复杂的正确性证明过程。。Dijkstra 的梦想最终并没有实现。没有几个程序员会认为形式化验证是产出高质量软件的必备条件。

当然，形式化的、欧几里得式的数学推导证明并不是证明结构化编程正确性的唯一手段。下面我们来看另外一个十分成功的策略：*科学证明法*。

科学来救场

科学和数学在证明方法上有着根本性的不同，科学理论和科学定律通常是无法被证明的，譬如我们并没有办法证明牛顿第二运动定律 $F=ma$ 或者万有引力定律 $F=Gm_1m_2/r^2$ 是正确的，但我们可以用实际案例来演示这些定律的正确性，并通过高

精度测量来证明当相关精度达到小数点后多少位时，被测量对象仍然一直满足这个定律。但我们始终没有办法像用数学方法一样推导出这个定律。而且，不管我们进行多少次正确的实验，也无法排除今后会存在某一次实验可以推翻牛顿第二运动定律与万有引力定律的可能性。

这就是科学理论和科学定律的特点：它们可以被证伪，但是没有办法被证明。

但是我们仍然每天都在依赖这些定律生活。开车的时候，我们就等于是在用性命担保 $F=ma$ 是对世界运转方式的一个可靠的描述。每当我们迈出一步的时候，就等于在亲身证明 $F=Gm_1m_2/r^2$ 是正确的。

科学方法论不需要证明某条结论是正确的，只需要想办法证明它是错误的。如果某个结论经过一定的努力无法证伪，我们则认为它在当下是足够正确的。

当然，不是所有的结论都可以被证明或者证伪的。举一个最简单的不可证明的例子：“这句话是假的”，非真也非伪。

最终，我们可以说数学是要将可证明的结论证明，而与之相反，科学研究则是要将可证明的结论证伪。

测试

Dijkstra 曾经说过“测试只能展示 Bug 的存在，并不能证明不存在 Bug”，换句话说，一段程序可以由一个测试来证明其错误性，但是却不能被证明是正确的。测试的作用是让我们得出某段程序已经足够实现当前目标这一结论。

这一事实所带来的影响是惊人的。软件开发虽然看起来是在操作很多数学结构，其实不是一个数学研究过程。恰恰相反，软件开发更像是一门科学研究学科，我们通过无法证伪来证明软件的正确性。

注意，这种证伪过程只能应用于可证明的程序上。某段程序如果是不可证明的，例如，其中采用了不加限制的 `goto` 语句，那么无论我们为它写多少测试，也不能够证明其正确性。

结构化编程范式促使我们先将一段程序递归降解为一系列可证明的小函数，然后再编写相关的测试来试图证明这些函数是错误的。如果这些测试无法证伪这些函数，那么我们就可以认为这些函数是足够正确的，进而推导整个程序是正确的。

本章小结

结构化编程范式中最有价值的地方就是，它赋予了我们创造可证伪程序单元的能力。这就是为什么现代编程语言一般不支持无限制的 goto 语句。更重要的是，这也是为什么在架构设计领域，功能性降解拆分仍然是最佳实践之一。

无论在哪一个层面上，从最小的函数到最大组件，软件开发的过程都和科学研究非常类似，它们都是由证伪驱动的。软件架构师需要定义可以方便地进行证伪（测试）的模块、组件以及服务。为了达到这个目的，他们需要将类似结构化编程的限制方法应用在更高的层面上。

我们在接下来的章节中将会深入研究这些限制性的方法。

第 5 章

面向对象编程

稍后我们会讲到，设计一个优秀的软件架构要基于对面向对象设计（Object-Oriented Design）的深入理解及应用。但我们首先得弄明白一个问题：究竟什么是面向对象？

对于这个问题，一种常见的回答是“数据与函数的组合”。这种说法虽然被广为引用，但总显得并不是那么贴切，因为它似乎暗示了 `o.f()` 与 `f(o)` 之间是有区别的，这显然不是事实。面向对象理论是在 1966 年提出的，当时 Dahl 和 Nygaard 主要是将函数调用栈迁移到了堆区域中。数据结构被用作函数的调用参数这件事情远比这发生的时间更早。

另一种常见的回答是“面向对象编程是一种对真实世界进行建模的方式”，这种回答只能算作避重就轻。“对真实世界的建模”到底要如何进行？我们为什么要这么做，有什么好处？也许这句话意味着是“由于采用面向对象方式构建的软件与真实世界的关系更紧密，所以面向对象编程可以使得软件开发更容易”——即使这样说，也仍然逃避了关键问题——面向对象编程究竟是什么？

还有些人在回答这个问题的时候，往往会搬出一些神秘的词语，譬如封装（*encapsulation*）、继承（*inheritance*）、多态（*polymorphism*）。其隐含意思就是说面向对象编程是这三项的有机组合，或者任何一种支持面向对象的编程语言必须支持这三个特性。

那么，我们接下来可以逐个来分析一下这三个概念。

封装

由于面向对象编程语言为我们方便而有效地封装数据和函数提供了有力的支持，导致封装这个概念经常被引用为面向对象编程定义的一部分。通过采用封装特性，我们可以把一组相关联的数据和函数圈起来，使圈外面的代码只能看见部分函数，数据则完全不可见。譬如，在实际应用中，类（class）中的公共函数和私有成员变量就是这样。

然而，这个特性其实并不是面向对象编程所独有的。其实，C 语言也支持完整

的封装，下面来看一个简单的 C 程序：

point.h

```
struct Point;
struct Point* makePoint(double x, double y);
double distance (struct Point *p1, struct Point *p2);
```

point.c

```
#include "point.h"
#include <stdlib.h>
#include <math.h>

struct Point {
  double x,y;
};

struct Point* makepoint(double x, double y) {
  struct Point* p = malloc(sizeof(struct Point));
  p->x = x;
  p->y = y;
  return p;
}

double distance(struct Point* p1, struct Point* p2) {
  double dx = p1->x - p2->x;
  double dy = p1->y - p2->y;
  return sqrt(dx*dx+dy*dy);
}
```

显然，使用 point.h 的程序是没有 Point 结构体成员的访问权限的。它们只能调用 makePoint() 函数和 distance() 函数，但对它们来说，Point 这个数据结构体的内部细节，以及函数的具体实现方式都是不可见的。

这正是完美封装——虽然 C 语言是非面向对象的编程语言。上述 C 程序是很常见的。在头文件中进行数据结构以及函数定义的前置声明（forward declare），然后在程序文件中具体实现。程序文件中的具体实现细节对使用者来说是不可见的。

而 C++作为一种面向对象编程语言，反而破坏了 C 的完美封装性。

由于一些技术原因[1]，C++编译器要求类的成员变量必须在该类的头文件中声明。这样一来，我们的 point.h 程序随之就改成了这样：

point.h

```
class Point {
public:
  Point(double x, double y);
  double distance(const Point& p) const;

private:
  double x;
  double y;
};
```

point.cc

```
#include "point.h"
#include <math.h>

Point::Point(double x, double y)
: x(x), y(y)
{}

double Point::distance(const Point& p) const {
  double dx = x-p.x;
  double dy = y-p.y;
  return sqrt(dx*dx + dy*dy);
}
```

好了，point.h 文件的使用者现在知道了成员变量 x 和 y 的存在！虽然编译器会禁止对这两个变量的直接访问，但是使用者仍然知道了它们的存在。而且，如果 x 和 y 变量名称被改变了，point.cc 也必须重新编译才行！这样的封装性显然是不完美的。

当然，C++通过在编程语言层面引入 public、private、protected 这些关键词，部分维护了封装性。但所有这些都是为了解决编译器自身的技术实现问题而

1　C++编译器必须要知道每个类实例的大小。

引入的 hack——编译器由于技术实现原因必须在头文件中看到成员变量的定义。

而 Java 和 C#则彻底抛弃了头文件与实现文件分离的编程方式，这其实进一步削弱了封装性。因为在这些语言中，我们是无法区分一个类的声明和定义的。

由于上述原因，我们很难说强封装是面向对象编程的必要条件。而事实上，有很多面向对象编程语言[1]对封装性并没有强制性的要求。

面向对象编程在应用上确实会要求程序员尽量避免破坏数据的封装性。但实际情况是，那些声称自己提供面向对象编程支持的编程语言，相对于 C 这种完美封装的语言而言，其封装性都被削弱了，而不是加强了。

继承

既然面向对象编程语言并没有提供更好的封装性，那么在继承性方面又如何呢？

嗯，其实也就一般般吧。简而言之，继承的主要作用是让我们可以在某个作用域内对外部定义的某一组变量与函数进行覆盖。这事实上也是 C 程序员[2]早在面向对象编程语言发明之前就一直在做的事了。

下面，看一下刚才的 C 程序 `point.h` 的扩展版：

namedPoint.h

```
struct NamedPoint;

struct NamedPoint* makeNamedPoint(double x, double y, char* name);
void setName(struct NamedPoint* np, char* name);
char* getName(struct NamedPoint* np);
```

namedPoint.c

```
#include "namedPoint.h"
```

1 例如 Smalltalk、Python、JavaScript、Lua、Ruby。

2 不仅是 C 语言，大部分同时期的编程语言都提供了将某种数据结构伪装成另外一种数据结构的特性。

```
#include <stdlib.h>

struct NamedPoint {
  double x,y;
  char* name;
};

struct NamedPoint* makeNamedPoint(double x, double y, char* name) {
  struct NamedPoint* p = malloc(sizeof(struct NamedPoint));
  p->x = x;
  p->y = y;
  p->name = name;
  return p;
}

void setName(struct NamedPoint* np, char* name) {
  np->name = name;
}

char* getName(struct NamedPoint* np) {
  return np->name;
}
```

main.c

```
#include "point.h"
#include "namedPoint.h"
#include <stdio.h>

int main(int ac, char** av) {
  struct NamedPoint* origin = makeNamedPoint(0.0, 0.0, "origin");
  struct NamedPoint* upperRight = makeNamedPoint
    (1.0, 1.0, "upperRight");
  printf("distance=%f\n",
    distance(
              (struct Point*) origin,
              (struct Point*) upperRight));
}
```

请仔细观察 main 函数，这里 NamedPoint 数据结构是被当作 Point 数据结

构的一个衍生体来使用的。之所以可以这样做，是因为 NamedPoint 结构体的前两个成员的顺序与 Point 结构体的完全一致。简单来说，NamedPoint 之所以可以被伪装成 Point 来使用，是因为 NamedPoint 是 Point 结构体的一个超集，同时两者共同成员的顺序也是一样的。

上面这种编程方式虽然看上去有些投机取巧，但是在面向对象理论被提出之前，这已经很常见了[1]。其实，C++内部就是这样实现单继承的。

因此，我们可以说，早在面向对象编程语言被发明之前，对继承性的支持就已经存在很久了。当然了，这种支持用了一些投机取巧的手段，并不像如今的继承这样便利易用，而且，多重继承（multiple inheritance）如果还想用这种方法来实现，就更难了。

同时应该注意的是，在 main.c 中，程序员必须强制将 NamedPoint 的参数类型转换为 Point，而在真正的面向对象编程语言中，这种类型的向上转换通常应该是隐性的。

综上所述，我们可以认为，虽然面向对象编程在继承性方面并没有开创出新，但是的确在数据结构的伪装性上提供了相当程度的便利性。

回顾一下到目前为止的分析，面向对象编程在封装性上得 0 分，在继承性上勉强可以得 0.5 分（满分为 1）。

下面，我们还有最后一个特性要讨论。

多态

在面向编程对象语言被发明之前，我们所使用的编程语言能支持多态吗？

答案是肯定的，请注意看下面这段用 C 语言编写的 copy 程序：

1 这种编程方式到现在也很常见。

```
#include <stdio.h>

void copy() {
   int c;
   while ((c=getchar()) != EOF)
     putchar(c);
}
```

在上述程序中，函数 getchar() 主要负责从 STDIN 中读取数据。但是 STDIN 究竟指代的是哪个设备呢？同样的道理，putchar() 主要负责将数据写入 STDOUT，而 STDOUT 又指代的是哪个设备呢？很显然，这类函数其实就具有多态性，因为它们的行为依赖于 STDIN 和 STDOUT 的具体类型。

这里的 STDIN 和 STDOUT 与 Java 中的接口类似，各种设备都有各自的实现。当然，这个 C 程序中是没有接口这个概念的，那么 getchar() 这个调用的动作是如何真正传递到设备驱动程序中，从而读取到具体内容的呢？

其实很简单，UNIX 操作系统强制要求每个 IO 设备都要提供 open、close、read、write 和 seek 这 5 个标准函数。[1]也就是说，每个 IO 设备驱动程序对这 5 种函数的实现在函数调用上必须保持一致。

首先，FILE 数据结构体中包含了相对应的 5 个函数指针，分别用于指向这些函数：

```
struct FILE {
  void (*open)(char* name, int mode);
  void (*close)();
  int (*read)();
  void (*write)(char);
  void (*seek)(long index, int mode);
};
```

然后，譬如控制台设备的 IO 驱动程序就会提供这 5 个函数的实际定义，将 FILE 结构体的函数指针指向这些对应的实现函数：

1　UNIX 系统有很多变体，这里只是举了一个例子。

```
#include "file.h"

void open(char* name, int mode) {/*...*/}
void close() {/*...*/}
int read() {int c;/*...*/ return c;}
void write(char c) {/*...*/}
void seek(long index, int mode) {/*...*/}

struct FILE console = {open, close, read, write, seek};
```

现在，如果 STDIN 的定义是 FILE*，并同时指向了 console 这个数据结构，那么 getchar()的实现方式就是这样的：

```
extern struct FILE* STDIN;

int getchar() {
  return STDIN->read();
}
```

换句话说，getchar()只是调用了 STDIN 所指向的 FILE 数据结构体中的 read 函数指针指向的函数。

这个简单的编程技巧正是面向对象编程中多态的基础。例如在 C++中，类中的每个虚函数（virtual function）的地址都被记录在一个名叫 vtable 的数据结构里。我们对虚函数的每次调用都要先查询这个表，其衍生类的构造函数负责将该衍生类的虚函数地址加载到整个对象的 vtable 中。

归根结底，多态其实不过就是函数指针的一种应用。自从 20 世纪 40 年代末期冯·诺依曼架构诞生那天起，程序员们就一直在使用函数指针模拟多态了。也就是说，面向对象编程在多态方面没有提出任何新概念。

当然了，面向对象编程语言虽然在多态上并没有理论创新，但它们也确实让多态变得更安全、更便于使用了。

用函数指针显式实现多态的问题就在于函数指针的危险性。毕竟，函数指针的调用依赖于一系列需要人为遵守的约定。程序员必须严格按照固定约定来初始化函数指针，并同样严格地按照约定来调用这些指针。只要有一个程序员没有遵守这些

约定，整个程序就会产生极其难以跟踪和消除的 Bug。

面向对象编程语言为我们消除了人工遵守这些约定的必要，也就等于消除了这方面的危险性。采用面向对象编程语言让多态实现变得非常简单，让一个传统 C 程序员可以去做以前不敢想的事情。综上所述，我们认为面向对象编程其实是对程序间接控制权的转移进行了约束。

多态的强大性

那么多态的优势在哪里呢？为了让读者更好地理解多态的好处，我们需要再来看一下刚才的 copy 程序。如果要支持新的 IO 设备，该程序需要做什么改动呢？譬如，假设我们想要用该 copy 程序从一个手写识别设备将数据复制到另一个语音合成设备中，我们需要针对 copy 程序做什么改动，才能实现这个目标呢？

答案是完全不需要做任何改动！确实，我们甚至不需要重新编译该 copy 程序。为什么？因为 copy 程序的源代码并不依赖于 IO 设备驱动程序的代码。只要 IO 设备驱动程序实现了 FILE 结构体中定义的 5 个标准函数，该 copy 程序就可以正常使用它们。

简单来说，IO 设备变成了 copy 程序的插件。

为什么 UNIX 操作系统会将 IO 设备设计成插件形式呢？因为自 20 世纪 50 年代末期以来，我们学到了一个重要经验：程序应该与设备无关。这个经验从何而来呢？因为一度所有程序都是设备相关的，但是后来我们发现自己其实真正需要的是在不同的设备上实现同样的功能。

例如，我们曾经写过一些程序，需要从卡片盒中的打孔卡片[1]读取数据，同时要通过在新的卡片上打孔来输出数据。后来，客户不再使用打孔卡片，而开始使用磁带卷了。这就给我们带来了很多麻烦，很多程序都需要重写。于是我们就会想，如果这段程序可以同时操作打孔卡片和磁带那该多好。

1　打孔卡片，即 IBM Hollerith 卡，80 格宽。虽然这种卡片在 20 世纪 50 年代、60 年代，甚至 70 年代都很常见，但是现在大部分人应该都没有见过了。

插件式架构就是为了支持这种 IO 不相关性而发明的，它几乎在随后的所有操作系统中都有应用。但即使多态有如此多优点，大部分程序员还是没有将插件特性引入他们自己的程序中，因为函数指针实在是太危险了。

而面向对象编程的出现使得这种插件式架构可以在任何地方被安全地使用。

依赖反转

我们可以想象一下在安全和便利的多态支持出现之前，软件是什么样子的。下面有一个典型的调用树的例子，`main` 函数调用了一些高层函数，这些高层函数又调用了一些中层函数，这些中层函数又继续调用了一些底层函数。在这里，源代码层面的依赖不可避免地要跟随程序的控制流（详见图 5.1）。

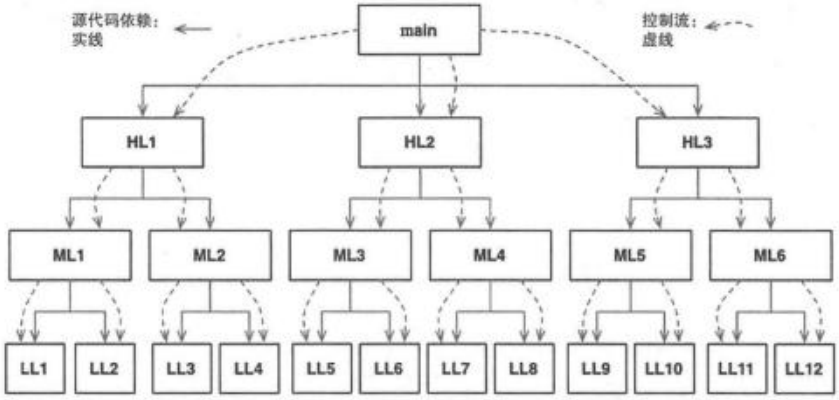


图 5.1：源代码依赖与控制流的区别

如你所见，`main` 函数为了调用高层函数，它就必须能够看到这个函数所在的模块。在 C 中，我们会通过`#include` 来实现，在 Java 中则通过 `import` 来实现，而在 C#中则用的是 `using` 语句。总之，每个函数的调用方都必须要引用被调用方所在的模块。

显然，这样做就导致了我们在软件架构上别无选择。在这里，系统行为决定了控制流，而控制流则决定了源代码依赖关系。

但一旦我们使用了多态，情况就不一样了（详见图 5.2）。

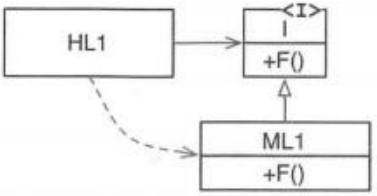


图 5.2：依赖反转

在图 5.2 中，模块 HL1 调用了 ML1 模块中的 F() 函数，这里的调用是通过源代码级别的接口来实现的。当然在程序实际运行时，接口这个概念是不存在的，HL1 会调用 ML1 中的 F() 函数[1]。

请注意模块 ML1 和接口 I 在源代码上的依赖关系（或者叫继承关系），该关系的方向和控制流正好是相反的，我们称之为*依赖反转*。这种反转对软件架构设计的影响是非常大的。

事实上，通过利用面向编程语言所提供的这种安全便利的多态实现，无论我们面对*怎样的源代码级别的依赖关系*，都可以*将其反转*。

现在，我们可以再回头来看图 5.1 中的调用树，就会发现其中的众多源代码依赖关系都可以通过引入接口的方式来进行反转。

通过这种方法，软件架构师可以*完全控制*采用了面向对象这种编程方式的系统中所有的源代码依赖关系，而不再受到系统控制流的限制。不管哪个模块调用或者被调用，软件架构师都可以随意更改源代码依赖关系。

这就是面向对象编程的好处，同时也是面向对象编程这种范式的核心本质——至少对一个软件架构师来说是这样的。

这种能力有什么用呢？在下面的例子中，我们可以用它来让数据库模块和用户界面模块都依赖于业务逻辑模块（见图 5.3），而非相反。

1　这种调用是隐性、间接进行的。

图 5.3：数据库和用户界面都依赖于业务逻辑

这意味着我们让用户界面和数据库都成为业务逻辑的插件。也就是说，业务逻辑模块的源代码不需要引入用户界面和数据库这两个模块。

这样一来，业务逻辑、用户界面以及数据库就可以被编译成三个独立的组件或者部署单元（例如 jar 文件、DLL 文件、Gem 文件等）了，这些组件或者部署单元的依赖关系与源代码的依赖关系是一致的，业务逻辑组件也不会依赖于用户界面和数据库这两个组件。

于是，业务逻辑组件就可以*独立于用户界面和数据库来进行部署*了，我们对用户界面或者数据库的修改将不会对业务逻辑产生任何影响，这些组件都可以被分别、独立地部署。

简单来说，当某个组件的源代码需要修改时，仅仅需要重新部署该组件，不需要更改其他组件，这就是*独立部署能力*。

如果系统中的所有组件都可以独立部署，那它们就可以由不同的团队并行开发，这就是所谓的*独立开发能力*。

本章小结

面向对象编程到底是什么？业界在这个问题上存在着很多不同的说法和意见。然而对一个软件架构师来说，其含义应该是非常明确的：面向对象编程就是以多态为手段来对源代码中的依赖关系进行控制的能力，这种能力让软件架构师可以构建出某种插件式架构，让高层策略性组件与底层实现性组件相分离，底层组件可以被编译成插件，实现独立于高层组件的开发和部署。

第6章

函数式编程

函数式编程所依赖的原理，在很多方面其实是早于编程本身出现的。因为函数式编程这种范式强烈依赖于 Alonzo Church 在 20 世纪 30 年代发明的 λ 演算。

整数平方

我们最好还是用一个例子来解释什么是函数式编程。请看下面的这个例子：这段代码想要输出前 25 个整数的平方值。

如果使用 Java 语言，代码如下：

```
public class Squint {
 public static void main(String args[]) {
   for (int i=0; i<25; i++)
     System.out.println(i*i);
  }
}
```

下面我们改用 Clojure 语言来写这个程序，Clojure 是 LISP 语言的一种衍生体，属于函数式编程语言。其代码如下：

```
(println (take 25 (map (fn [x] (* x x)) (range))))
```

如果读者对 LISP 不熟悉，这段代码可能看起来很奇怪。没关系，让我们换一种格式，用注释来说明一下吧：

```
(println ;_____________________输出
  (take 25 ;___________________前 25
    (map (fn [x] (* x x)) ;_求平方
      (range)))) ;____________整数
```

很明显，这里的 println、take、map 和 range 都是函数。在 LISP 中，函数是通过括号来调用的，例如 (range) 表达式就是在调用 range 函数。

而表达式 (fn [x] (* x x)) 则是一个匿名函数，该函数用同样的值作为参数调用了乘法函数。换句话说，该函数计算的是平方值。

现在让我们回过头来再看一下这整句代码，从最内侧的函数调用开始：

- range 函数会返回一个从 0 开始的整数无穷列表。
- 然后该列表会被传入 map 函数，并针对列表中的每个元素，调用求平方值的匿名函数，产生了一个无穷多的、包含平方值的列表。
- 接着再将这个列表传入 take 函数，后者会返回一个仅包含前 25 个元素的新列表。
- println 函数将它的参数输出，该参数就是上面这个包含了 25 个平方值的列表。

读者不用担心上面提到的无穷列表。因为这些列表中的元素只有在被访问时才会被创建，所以实际上只有前 25 个元素是真正被创建了的。

如果上述内容还是让读者觉得云里雾里的话，可以自行学习一下 Clojure 和函数式编程，本书的目标并不是要教你学会这门语言，因此不再展开。

相反，我们讨论它的主要目标是要突显出 Clojure 和 Java 这两种语言之间的巨大区别。在 Java 程序中，我们使用的是可变量，即变量 i，该变量的值会随着程序执行的过程而改变，故被称为循环控制变量。而 Clojure 程序中是不存在这种变量的，变量 x 一旦被初始化之后，就不会再被更改了。

这句话有点出人意料：函数式编程语言中的变量（Variable）是不可变（Vary）的。

不可变性与软件架构

为什么不可变性是软件架构设计需要考虑的重点呢？为什么软件架构师要操心变量的可变性呢？答案显而易见：所有的竞争问题、死锁问题、并发更新问题都是由可变变量导致的。如果变量永远不会被更改，那就不可能产生竞争或者并发更新问题。如果锁状态是不可变的，那就永远不会产生死锁问题。

换句话说，一切并发应用遇到的问题，一切由于使用多线程、多处理器而引起的问题，如果没有可变变量的话都不可能发生。

作为一个软件架构师，当然应该要对并发问题保持高度关注。我们需要确保自己设计的系统在多线程、多处理器环境中能稳定工作。所以在这里，我们实际应该要问的问题是：不可变性是否实际可行？

如果我们能忽略存储器与处理器在速度上的限制，那么答案是肯定的。否则的话，不可变性只有在一定情况下是可行的。

下面让我们来看一下它具体该如何做到可行。

可变性的隔离

一种常见方式是将应用程序，或者是应用程序的内部服务进行切分，划分为可变的和不可变的两种组件。不可变组件用纯函数的方式来执行任务，期间不更改任何状态。这些不可变的组件将通过与一个或多个非函数式组件通信的方式来修改变量状态（参见图 6.1）。

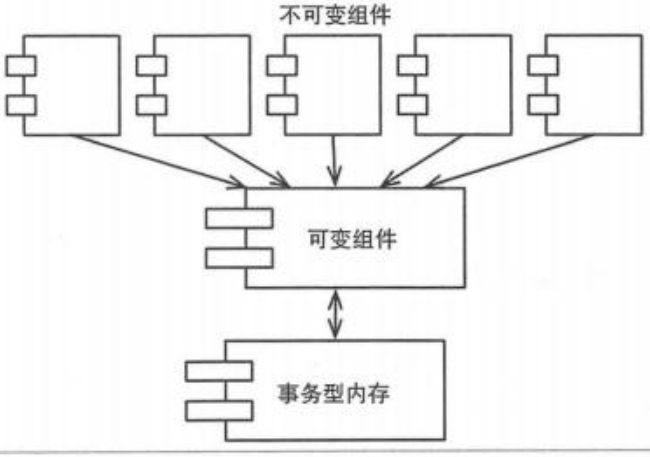


图 6.1：修改状态以及事务型内存

由于状态的修改会导致一系列并发问题的产生，所以我们通常会采用某种事务型内存来保护可变变量，避免同步更新和竞争状态的发生。

事务型内存基本上与数据库保护磁盘数据的方式[1]类似，通常采用的是事务或者重试机制。

下面我们可以用 Clojure 中的 atom 机制来写一个简单的例子：

```
(def counter (atom 0)) ;   初始化计数器为 0
(swap! counter inc) ;      安全地进行计数器递增
```

在这段代码中，counter 变量被定义为 atom 类型。在 Clojure 中，atom 是一类特殊的变量，它被允许在 swap!函数定义的严格条件下进行更改。

至于 swap!函数，如同上面代码所写，它需要两个参数：一个是被用来修改的 atom 类型实例，另一个是用来计算新值的函数。在上面的代码中，inc 函数会将参数加 1 并存入 counter 这个 atom 实例。

在这里，swap!所采用的策略是传统的比较+替换算法。即先读取 counter 变量的值，再将其传入 inc 函数。然后当 inc 函数返回时，将原先用锁保护起来的 counter 值与传入 inc 时的值进行比较。如果两边的值一致，则将 inc 函数返回的值存入 counter，释放锁。否则，先释放锁，再从头进行重试。

当然，atom 这个机制只适用于上面这种简单的应用程序，它并不适用于解决由多个相关变量同时需要更改所引发的并发更新问题和死锁问题，要想解决这些问题，我们就需要用到更复杂的机制。

这里的要点是：一个架构设计良好的应用程序应该将状态修改的部分和不需要修改状态的部分隔离成单独的组件，然后用合适的机制来保护可变量。

软件架构师应该着力于将大部分处理逻辑都归于不可变组件中，可变状态组件的逻辑应该越少越好。

事件溯源

随着存储和处理能力的大幅进步，现在拥有每秒可以执行数十亿条指令的处理

1　当然，现在谁还在用磁盘呢？

器，以及数十亿字节内存的计算机已经很常见了。而内存越大，处理速度越快，我们对可变状态的依赖就会越少。

举个简单的例子，假设某个银行应用程序需要维护客户账户余额信息，当它执行存取款事务时，就要同时负责修改余额记录。

如果我们不保存具体账户余额，仅仅保存事务日志，那么当有人想查询账户余额时，我们就将全部交易记录取出，并且每次都得从最开始到当下进行累计。当然，这样的设计就不需要维护任何可变变量了。

但显而易见，这种实现是有些不合理的。因为随着时间的推移，事务的数目会无限制增长，每次处理总额所需要的处理能力很快就会变得不能接受。如果想使这种设计永远可行的话，我们将需要无限容量的存储，以及无限的处理能力。

但是可能我们并不需要这个设计永远可行，而且可能在整个程序的生命周期内，我们有足够的存储和处理能力来满足它。

这就是*事件溯源*[1]，在这种体系下，我们只存储事务记录，不存储具体状态。当需要具体状态时，我们只要从头开始计算所有的事务即可。

在存储方面，这种架构的确需要很大的存储容量。如今离线数据存储器的增长是非常快的，现在 1 TB 对我们来说也已经不算什么了。

更重要的是，这种数据存储模式中不存在删除和更新的情况，我们的应用程序不是 CRUD，而是 CR。因为更新和删除这两种操作都不存在了，自然也就不存在并发问题。

如果我们有足够大的存储量和处理能力，应用程序就可以用完全不可变的、*纯函数式*的方式来编程。

如果读者还是觉得这听起来不太靠谱，可以想想我们现在用的源代码管理程序，它们正是用这种方式工作的！

1 感谢 Greg Young 在这方面的指导。

本章小结

下面我们来总结一下：

- 结构化编程是对程序控制权的直接转移的限制。
- 面向对象编程是对程序控制权的间接转移的限制。
- 函数式编程是对程序中赋值操作的限制。

这三个编程范式都对程序员提出了新的限制。每个范式都约束了某种编写代码的方式，没有一个编程范式是在增加新能力。

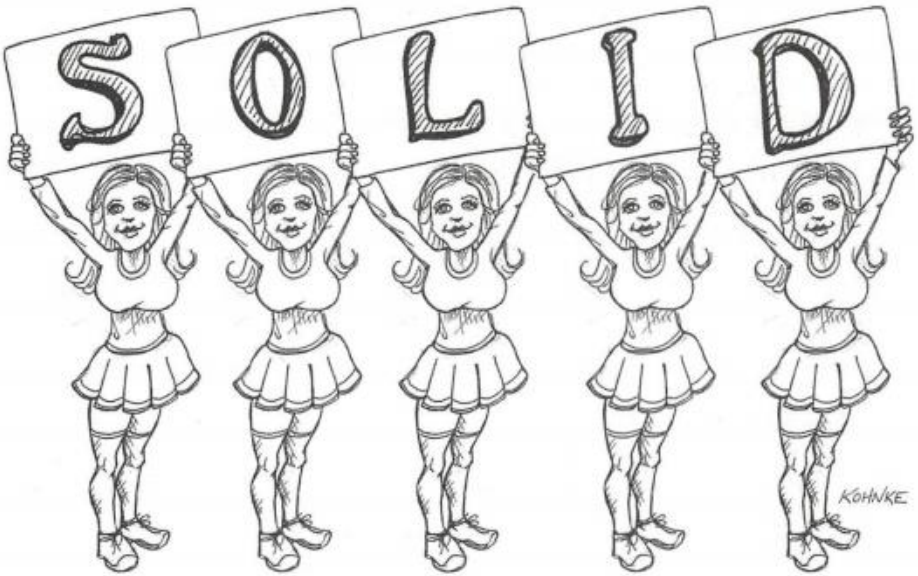
也就是说，我们过去 50 年学到的东西主要是——什么不应该做。

我们必须面对这种不友好的现实：软件构建并不是一个迅速前进的技术。今天构建软件的规则和 1946 年阿兰·图灵写下电子计算机的第一行代码时是一样的。尽管工具变化了，硬件变化了，但是软件编程的核心没有变。

总而言之，软件，或者说计算机程序无一例外是由顺序结构、分支结构、循环结构和间接转移这几种行为组合而成的，无可增加，也缺一不可。

第 3 部分

设计原则



通常来说，要想构建一个好的软件系统，应该从写整洁的代码开始做起。毕竟，如果建筑所使用的砖头质量不佳，那么架构所能起到的作用也会很有限。反之亦然，如果建筑的架构设计不佳，那么其所用的砖头质量再好也没有用。这就是 SOLID 设计原则所要解决的问题。

SOLID 原则的主要作用就是告诉我们如何将数据和函数组织成为类，以及如何将这些类链接起来成为程序。请注意，这里虽然用到了“类”这个词，但是并不意味着我们将要讨论的这些设计原则仅仅适用于面向对象编程。这里的类仅仅代表了一种数据和函数的分组，每个软件系统都会有自己的分类系统，不管它们各自是不是将其称为“类”，事实上都是 SOLID 原则的适用领域。

一般情况下，我们为软件构建中层结构的主要目标如下：

- 使软件可容忍被改动。
- 使软件更容易被理解。
- 构建可在多个软件系统中复用的组件。

我们在这里之所以会使用“中层”这个词，是因为这些设计原则主要适用于那些进行模块级编程的程序员。SOLID 原则应该直接紧贴于具体的代码逻辑之上，这些原则是用来帮助我们定义软件架构中的组件和模块的。

当然了，正如用好砖也会盖歪楼一样，采用设计良好的中层组件并不能保证系统的整体架构运作良好。正因为如此，我们在讲完 SOLID 原则之后，还会再继续针对组件的设计原则进行更进一步的讨论，将其推进到高级软件架构部分。

SOLID 原则的历史已经很悠久了，早在 20 世纪 80 年代末期，我在 USENET 新闻组（该新闻组在当时就相当于今天的 Facebook）上和其他人辩论软件设计理念的时候，该设计原则就已经开始逐渐成型了。随着时间的推移，其中有一些原则得到了修改，有一些则被抛弃了，还有一些被合并了，另外也增加了一些。它们的最终形态是在 2000 年左右形成的，只不过当时采用的是另外一个展现顺序。

2004 年前后，Michael Feathers 的一封电子邮件提醒我：如果重新排列这些设计原则，那么它们的首字母可以排列成 SOLID——这就是 SOLID 原则诞生的故事。

在这一部分中，我们会逐章地详细讨论每个设计原则，下面先来做一个简单摘要。

- SRP：单一职责原则。

该设计原则是基于康威定律（Conway's Law）[1]的一个推论——一个软件系统的最佳结构高度依赖于开发这个系统的组织的内部结构。这样，每个软件模块都有且只有一个需要被改变的理由。

- OCP：开闭原则。
  该设计原则是由 Bertrand Meyer 在 20 世纪 80 年代大力推广的，其核心要素是：如果软件系统想要更容易被改变，那么其设计就必须允许新增代码来修改系统行为，而非只能靠修改原来的代码。
- LSP：里氏替换原则。
  该设计原则是 Barbara Liskov 在 1988 年提出的一个著名的子类型定义。简单来说，这项原则的意思是如果想用可替换的组件来构建软件系统，那么这些组件就必须遵守同一个约定，以便让这些组件可以相互替换。
- ISP：接口隔离原则。
  这项设计原则主要告诫软件设计师应该在设计中避免不必要的依赖。
- DIP：依赖反转原则。
  该设计原则指出高层策略性的代码不应该依赖实现底层细节的代码，恰恰相反，那些实现底层细节的代码应该依赖高层策略性的代码。

这些年来，这些设计原则在很多不同的出版物中都有过详细描述[2]。在接下来的章节中，我们将会主要关注这些原则在软件架构上的意义，而不再重复其细节信息。如果你对这些原则并不是特别了解，那么我建议你先通过脚注中的文档熟悉一下它们，否则接下来的章节可能有点难以理解。

1　Melvin Conway 于 20 世纪 60 年代后期提出，任意一个软件都能反映出其制作团队的组织结构，这是因为人们会以反映他们组织形式的方式工作。换句话说，分散的团队可能用分散的架构生成系统。项目团队的组织结构中的优点和弱点都将不可避免地反映在他们生成的结果系统中。——编者注

2　例如 *Agile Software Development, Principles, Patterns, and Practices,* Robert C. Martin, Prentice Hall, 2002, http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod，以及 https://en.wikipedia. org/wiki/SOLID（或者 Google 搜索 SOLID 即可）。

第 7 章

SRP：单一职责原则

SRP 是 SOLID 五大设计原则中最容易被误解的一个。也许是名字的原因，很多程序员根据 SRP 这个名字想当然地认为这个原则就是指：每个模块都应该只做一件事。

没错，后者的确也是一个设计原则，即确保一个函数只完成一个功能。我们在将大型函数重构成小函数时经常会用到这个原则，但这只是一个面向底层实现细节的设计原则，并不是 SRP 的全部。

在历史上，我们曾经这样描述 SRP 这一设计原则：

> 任何一个软件模块都应该有且仅有一个被修改的原因。

在现实环境中，软件系统为了满足用户和所有者的要求，必然要经常做出这样那样的修改。而该系统的用户或者所有者就是该设计原则中所指的“被修改的原因”。所以，我们也可以这样描述 SRP：

> 任何一个软件模块都应该只对一个用户（User）或系统利益相关者（Stakeholder）负责。

不过，这里的“用户”和“系统利益相关者”在用词上也并不完全准确，它们很有可能指的是一个或多个用户和利益相关者，只要这些人希望对系统进行的变更是相似的，就可以归为一类——一个或多个有共同需求的人。在这里，我们将其称为行为者（*actor*）。

所以，对于 SRP 的最终描述就变成了：

> 任何一个软件模块都应该只对某一类行为者负责。

那么，上文中提到的“软件模块”究竟又是在指什么呢？大部分情况下，其最简单的定义就是指一个源代码文件。然而，有些编程语言和编程环境并不是用源代码文件来存储程序的。在这些情况下，“软件模块”指的就是一组紧密相关的函数和数据结构。

在这里，“相关”这个词实际上就隐含了 SRP 这一原则。代码与数据就是靠着与某一类行为者的相关性被组合在一起的。

或许，理解这个设计原则最好的办法就是让大家来看一些反面案例。反面案例1：重复的假象

这是我最喜欢举的一个例子：某个工资管理程序中的 Employee 类有三个函数 calculatePay()、reportHours()和 save()（见图 7.1）。

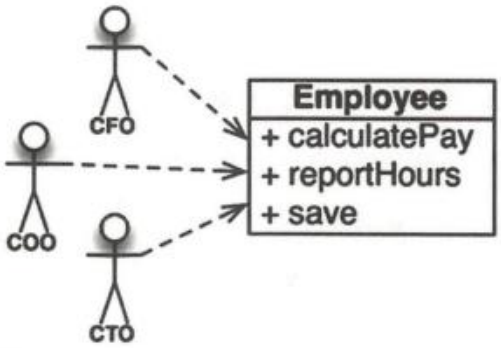


图 7.1：Employee 类

如你所见，这个类的三个函数分别对应的是三类非常不同的行为者，违反了 SRP 设计原则。

- calculatePay()函数是由财务部门制定的，他们负责向 CFO 汇报。
- reportHours()函数是由人力资源部门制定并使用的，他们负责向 COO 汇报。
- save()函数是由 DBA 制定的，他们负责向 CTO 汇报。

这三个函数被放在同一个源代码文件，即同一个 Employee 类中，程序员这样做实际上就等于使三类行为者的行为耦合在了一起，这有可能会导致 CFO 团队的命令影响到 COO 团队所依赖的功能。

例如，calculatePay()函数和 reportHours()函数使用同样的逻辑来计算正常工作时数。程序员为了避免重复编码，通常会将该算法单独实现为一个名为 regularHours()的函数（见图 7.2）。

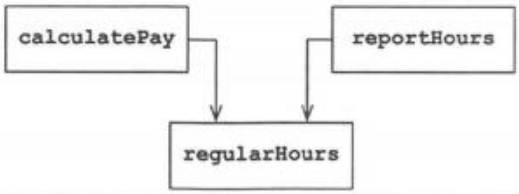


图 7.2：算法共享

接下来，假设 CFO 团队需要修改正常工作时数的计算方法，而 COO 带领的 HR 团队不需要这个修改，因为他们对数据的用法是不同的。

这时候，负责这项修改的程序员会注意到 `calculatePay()` 函数调用了 `regularHours()` 函数，但可能不会注意到该函数会同时被 `reportHours()` 调用。

于是，该程序员就这样按照要求进行了修改，同时 CFO 团队的成员验证了新算法工作正常。这项修改最终被成功部署上线了。

但是，COO 团队显然完全不知道这些事情的发生，HR 仍然在使用 `reportHours()` 产生的报表，随后就会发现他们的数据出错了！最终这个问题让 COO 十分愤怒，因为这些错误的数据给公司造成了几百万美元的损失。

与此类似的事情我们肯定多多少少都经历过。这类问题发生的根源就是因为我们将不同行为者所依赖的代码强凑到了一起。对此，SRP 强调*这类代码一定要被分开*。

反面案例 2：代码合并

一个拥有很多函数的源代码文件必然会经历很多次代码合并，该文件中的这些函数分别服务不同行为者的情况就更常见了。

例如，CTO 团队的 DBA 决定要对 `Employee` 数据库表结构进行简单修改。与此同时，COO 团队的 HR 需要修改工作时数报表的格式。

这样一来，就很可能出现两个来自不同团队的程序员分别对 `Employee` 类进行修改的情况。不出意外的话，他们各自的修改一定会互相冲突，这就必须要进行代码合并。

在这个例子中，这次代码合并不仅有可能让 CTO 和 COO 要求的功能出错，甚至连 CFO 原本正常的功能也可能受到影响。

事实上，这样的案例还有很多，我们就不一一列举了。它们的一个共同点是，多人为了不同的目的修改了同一份源代码，这很容易造成问题的产生。

而避免这种问题产生的方法就是将服务不同行为者的代码进行切分。

解决方案

我们有很多不同的方法可以用来解决上面的问题，每一种方法都需要将相关的函数划分成不同的类。

其中，最简单直接的办法是将数据与函数分离，设计三个类共同使用一个不包括函数的、十分简单的 `EmployeeData` 类（见图 7.3），每个类只包含与之相关的函数代码，互相不可见，这样就不存在互相依赖的情况了。

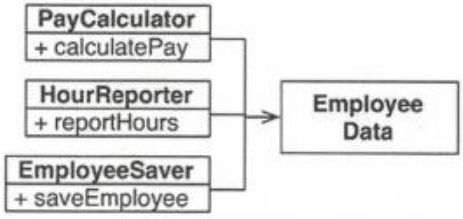


图 7.3：三个类互相不可见

这种解决方案的坏处在于：程序员现在需要在程序里处理三个类。另一种解决办法是使用 Facade 设计模式（见图 7.4）。

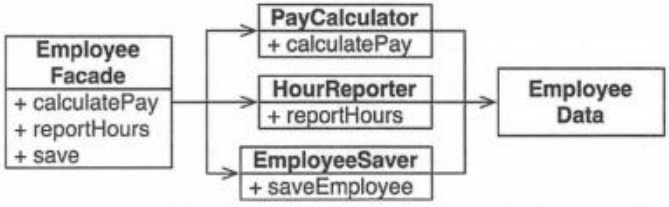


图 7.4：`Facade` 模式

这样一来，EmployeeFacade 类所需要的代码量就很少了，它仅仅包含了初始化和调用三个具体实现类的函数。

当然，也有些程序员更倾向于把最重要的业务逻辑与数据放在一起，那么我们也可以选择将最重要的函数保留在 Employee 类中，同时用这个类来调用其他没那么重要的函数（见图 7.5）。

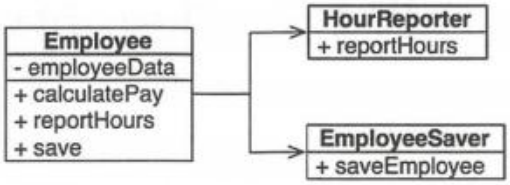


图 7.5：将最重要的函数保留在 Employee 类中，同时调用其他两个没那么重要的类

读者也许会反对上面这些解决方案，因为看上去这里的每个类中都只有一个函数。事实上并非如此，因为无论是计算工资、生成报表还是保存数据都是一个很复杂的过程，每个类都可能包含了许多私有函数。

总而言之，上面的每一个类都分别容纳了一组作用于相同作用域的函数，而在该作用域之外，它们各自的私有函数是互相不可见的。

本章小结

单一职责原则主要讨论的是函数和类之间的关系——但是它在两个讨论层面上会以不同的形式出现。在组件层面，我们可以将其称为共同闭包原则（Common Closure Principle），在软件架构层面，它则是用于奠定架构边界的变更轴心（Axis of Change）。我们在接下来的章节中会深入学习这些原则。

第 8 章

OCP：开闭原则

开闭原则（OCP）是 Bertrand Meyer 在 1988 年提出[1]的，该设计原则认为：

> 设计良好的计算机软件应该易于扩展，同时抗拒修改。

换句话说，一个设计良好的计算机系统应该在不需要修改的前提下就可以轻易被扩展。

其实这也是我们研究软件架构的根本目的。如果对原始需求的小小延伸就需要对原有的软件系统进行大幅修改，那么这个系统的架构设计显然是失败的。

尽管大部分软件设计师都已经认可了 OCP 是设计类与模块时的重要原则，但是在软件架构层面，这项原则的意义则更为重大。

下面，让我们用一个思想实验来做一些说明。

思想实验

假设我们现在要设计一个在 Web 页面上展示财务数据的系统，页面上的数据要可以滚动显示，其中负值应显示为红色。

接下来，该系统的所有者又要求同样的数据需要形成一个报表，该报表要能用黑白打印机打印，并且其报表格式要得到合理分页，每页都要包含页头、页尾及栏目名。同时，负值应该以括号表示。

显然，我们需要增加一些代码来完成这个要求。但在这里我们更关注的问题是，满足新的要求需要更改多少旧代码。

一个好的软件架构设计师会努力将旧代码的修改需求量降至最小，甚至为 0。

但该如何实现这一点呢？我们可以先将满足不同需求的代码分组（即 SRP），然后再来调整这些分组之间的依赖关系（即 DIP）。

1　请参考 Bertrand Meyer 在 1988 年发表的，由 Prentice Hall 出版的 *Object Oriented Software Construction* 一书的第 23 页。

利用 SRP，我们可以按图 8.1 中所展示的方式来处理数据流。即先用一段分析程序处理原始的财务数据，以形成报表的数据结构，最后再用两个不同的报表生成器来产生报表。

图 8.1： SRP 的应用

这里的核心就是将应用生成报表的过程拆成两个不同的操作。即先计算出报表数据，再生成具体的展示报表（分别以网页及纸质的形式展示）。

接下来，我们就该修改其源代码之间的依赖关系了。这样做的目的是保证其中一个操作被修改之后不会影响到另外一个操作。同时，我们所构建的新的组织形式应该保证该程序后续在行为上的扩展都无须修改现有代码。

在具体实现上，我们会将整个程序进程划分成一系列的类，然后再将这些类分割成不同的组件。下面，我们用图 8.2 中的那些双线框来具体描述一下整个实现。在这个图中，左上角的组件是 Controller，右上角是 Interactor，右下角是 Database，左下角则有四个组件分别用于代表不同的 Presenter 和 View。

在图 8.2 中，用<I>标记的类代表接口，用<DS>标记的则代表数据结构；开放箭头指代的是使用关系，闭合箭头则指代了实现与继承关系。

首先，我们在图 8.2 中看到的所有依赖关系都是其源代码中存在的依赖关系。这里，从类 A 指向类 B 的箭头意味着 A 的源代码中涉及了 B，但是 B 的源代码中并不涉及 A。因此在图 8.2 中，`FinancialDataMapper` 在实现接口时需要知道 `FinancialDataGateway` 的实现，而 `FinancialDataGateway` 则完全不必知道 `FinancialDataMapper` 的实现。

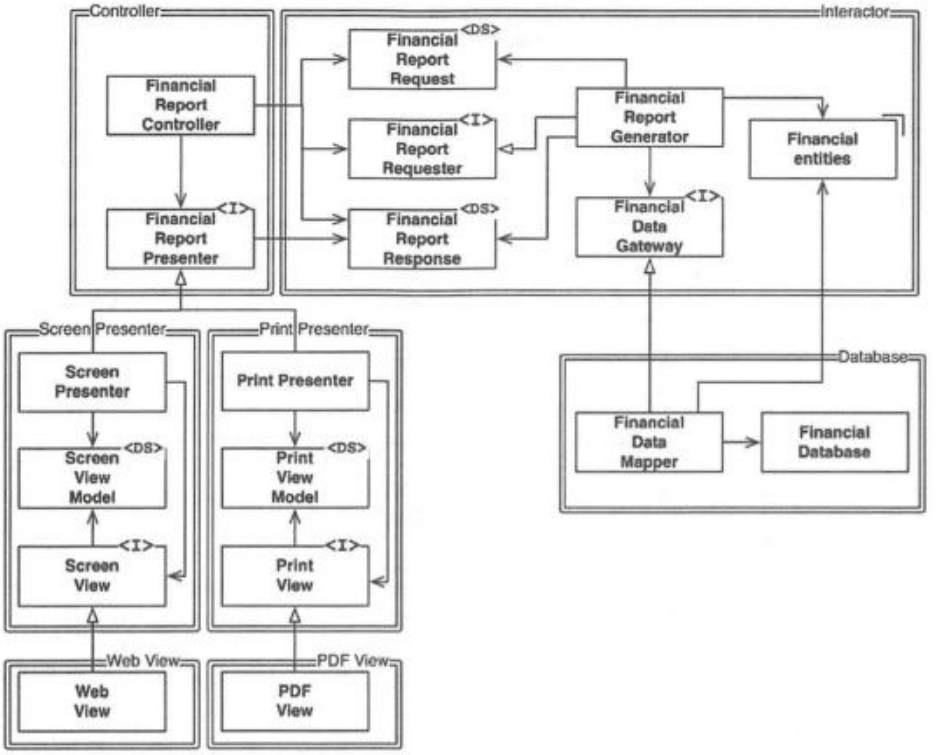


图 8.2：先将不同的操作划分为类，再将这些类分割为不同的组件

其次，这里很重要的一点是这些双线框的边界都是单向跨越的。也就是说，上图中所有组件之间的关系都是单向依赖的，如图 8.3 所示，图中的箭头都指向那些我们不想经常更改的组件。

让我们再来复述一下这里的设计原则：如果 A 组件不想被 B 组件上发生的修改所影响，那么就应该让 B 组件依赖于 A 组件。

所以现在的情况是，我们不想让发生在 Presenter 上的修改影响到 Controller，也不想让发生在 View 上的修改影响到 Presenter。而最关键的是，我们不想让任何修改影响到 Interactor。

其中，Interactor 组件是整个系统中最符合 OCP 的。发生在 Database、Controller、Presenter 甚至 View 上的修改都不会影响到 Interactor。

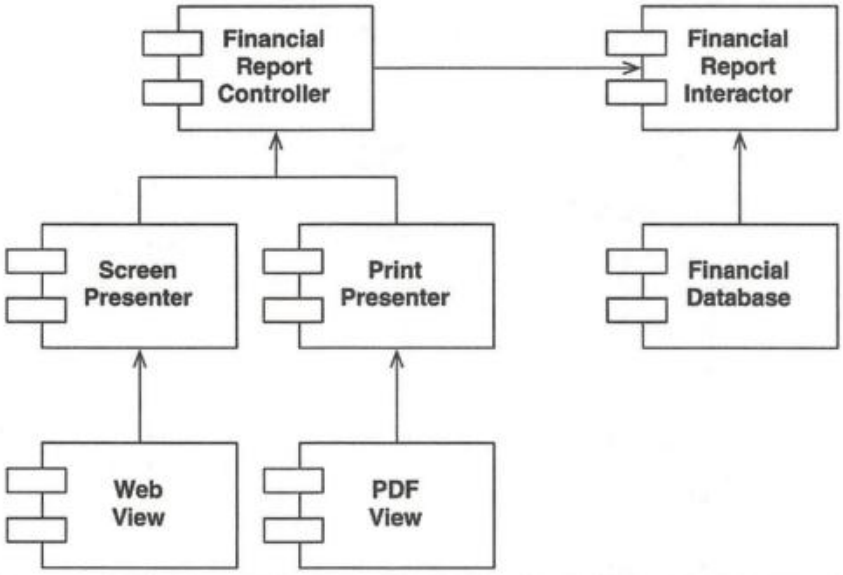


图 8.3：组件之间的关系都是单向的

为什么 Interactor 会被放在这么重要的位置上呢？因为它是该程序的业务逻辑所在之处，Interactor 中包含了其最高层次的应用策略。其他组件都只是负责处理周边的辅助逻辑，只有 Interactor 才是核心组件。

虽然 Controller 组件只是 Interactor 的附属品，但它却是 Presenter 和 View 所服务的核心。同样的，虽然 Presenter 组件是 Controller 的附属品，但它却是 View 所服务的核心。

另外需要注意的是，这里利用“层级”这个概念创造了一系列不同的保护层级。譬如，Interactor 是最高层的抽象，所以它被保护得最严密，而 Presenter 比 View 的层级高，但比 Controller 和 Interactor 的层级低。

以上就是我们在软件架构层次上对 OCP 这一设计原则的应用。软件架构师可以根据相关函数被修改的原因、修改的方式及修改的时间来对其进行分组隔离，并将这些互相隔离的函数分组整理成组件结构，使得高阶组件不会因低阶组件被修改而受到影响。

依赖方向的控制

如果刚刚的类设计把你吓着了，别害怕！你刚刚在图表中所看到的复杂度是我们想要对组件之间的依赖方向进行控制而产生的。

例如，`FinancialReportGenerator` 和 `FinancialDataMapper` 之间的 `FinancialDataGateway` 接口是为了反转 Interactor 与 Database 之间的依赖关系而产生的。同样的，`FinancialReportPresenter` 接口与两个 View 接口之间也类似于这种情况。

信息隐藏

当然，`FinancialReportRequester` 接口的作用则完全不同，它的作用是保护 `FinancialReportController` 不过度依赖于 Interactor 的内部细节。如果没有这个接口，则 Controller 将会传递性地依赖于 `FinancialEntities`。

这种传递性依赖违反了“软件系统不应该依赖其不直接使用的组件”这一基本原则。之后，我们会在讨论接口隔离原则和共同复用原则的时候再次提到这一点。

所以，虽然我们的首要目的是为了让 Interactor 屏蔽掉发生在 Controller 上的修改，但也需要通过隐藏 Interactor 内部细节的方法来让其屏蔽掉来自 Controller 的依赖。

本章小结

OCP 是我们进行系统架构设计的主导原则，其主要目标是让系统易于扩展，同时限制其每次被修改所影响的范围。实现方式是通过将系统划分为一系列组件，并且将这些组件间的依赖关系按层次结构进行组织，使得高阶组件不会因低阶组件被修改而受到影响。

第 9 章

LSP：里氏替换原则

1988 年，Barbara Liskov 在描述如何定义子类型时写下了这样一段话：

> 这里需要的是一种可替换性：如果对于每个类型是 S 的对象 o1 都存在一个类型为 T 的对象 o2，能使操作 T 类型的程序 P 在用 o2 替换 o1 时行为保持不变，我们就可以将 S 称为 T 的子类型。[1]

为了让读者理解这段话中所体现的设计理念，也就是里氏替换原则（LSP），我们可以来看几个例子。

继承的使用指导

假设我们有一个 `License` 类，其结构如图 9.1 所示。该类中有一个名为 `calcFee()` 的方法，该方法将由 `Billing` 应用程序来调用。而 `License` 类有两个“子类型”：`PersonalLicense` 与 `BusinessLicense`，这两个类会用不同的算法来计算授权费用。

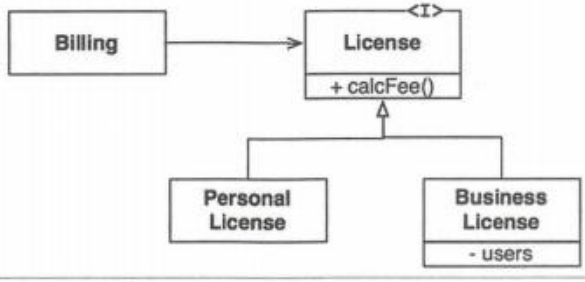


图 9.1：`License` 类与其衍生类，体现了 LSP 原则

上述设计是符合 LSP 原则的，因为 `Billing` 应用程序的行为并不依赖于其使用的任何一个衍生类。也就是说，这两个衍生类的对象都是可以用来替换 `License` 类对象的。

1　请参考 Barbara Liskov 的 *Data Abstraction and Hierarchy* 一文，该文 1988 年 5 月发表于 *SIGPLAN Notices*(23, 5)。

正方形/长方形问题

正方形/长方形问题是一个著名（或者说臭名远扬）的违反 LSP 的设计案例（该问题的结构如图 9.2 所示）。

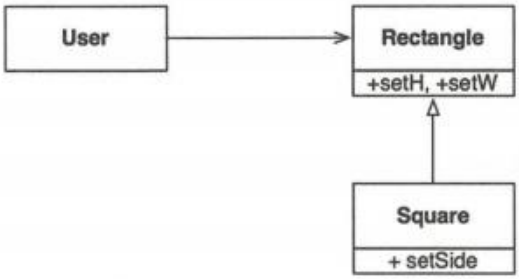


图 9.2：正方形/长方形问题

在这个案例中，Square 类并不是 Rectangle 类的子类型，因为 Rectangle 类的高和宽可以分别修改，而 Square 类的高和宽则必须一同修改。由于 User 类始终认为自己在操作 Rectangle 类，因此会带来一些混淆。例如在下面的代码中：

```
Rectangle r = …
r.setW(5);
r.setH(2);
assert(r.area() == 10);
```

很显然，如果上述代码在...处返回的是 Square 类，则最后的这个 assert 是不会成立的。

如果想要防范这种违反 LSP 的行为，唯一的办法就是在 User 类中增加用于区分 Rectangle 和 Square 的检测逻辑（例如增加 if 语句）。但这样一来，User 类的行为又将依赖于它所使用的类，这两个类就不能互相替换了。

LSP 与软件架构

在面向对象这场编程革命兴起的早期，我们的普遍认知正如上文所说，认为 LSP

只不过是指导如何使用继承关系的一种方法，然而随着时间的推移，LSP 逐渐演变成了一种更广泛的、指导接口与其实现方式的设计原则。

这里提到的接口可以有多种形式——可以是 Java 风格的接口，具有多个实现类；也可以像 Ruby 一样，几个类共用一样的方法签名，甚至可以是几个服务响应同一个 REST 接口。

LSP 适用于上述所有的应用场景，因为这些场景中的用户都依赖于一种接口，并且都期待实现该接口的类之间能具有可替换性。

想要从软件架构的角度来理解 LSP 的意义，最好的办法还是来看几个反面案例。

违反 LSP 的案例

假设我们现在正在构建一个提供出租车调度服务的系统。在该系统中，用户可以通过访问我们的网站，从多个出租车公司内寻找最适合自己的出租车。当用户选定车子时，该系统会通过调用 restful 服务接口来调度这辆车。

接下来，我们再假设该 restful 调度服务接口的 URI 被存储在司机数据库中。一旦该系统选中了最合适的出租车司机，它就会从司机数据库的记录中读取相应的 URI 信息，并通过调用这个 URI 来调度汽车。

也就是说，如果司机 Bob 的记录中包含如下调度 URI：

```
purplecab.com/driver/Bob
```

那么，我们的系统就会将调度信息附加在这个 URI 上，并发送这样一个 PUT 请求：

```
purplecab.com/driver/Bob
       /pickupAddress/24 Maple St.
       /pickupTime/153
       /destination/ORD
```

很显然，这意味着所有参与该调度服务的公司都必须遵守同样的 REST 接口，它们必须用同样的方式处理 pickupAddress、pickupTime 和 destination 字段。

接下来，我们再假设 Acme 出租车公司现在招聘的程序员由于没有仔细阅读上述接口定义，结果将 destination 字段缩写成了 dest。而 Acme 又是本地最大的出租车公司，另外，Acme CEO 的前妻不巧还是我们 CEO 的新欢……你懂的！这会对系统的架构造成什么影响呢？

显然，我们需要为系统增加一类特殊用例，以应对 Acme 司机的调度请求。而这必须要用另外一套规则来构建。

最简单的做法当然是增加一条 if 语句：

```
if (driver.getDispatchUri().startsWith("acme.com"))…
```

然而很明显，任何一个称职的软件架构师都不会允许这样一条语句出现在自己的系统中。因为直接将“acme”这样的字串写入代码会留下各种各样神奇又可怕的错误隐患，甚至会导致安全问题。

例如，Acme 也许会变得更加成功，最终收购了 Purple 出租车公司。然后，它们在保留了各自名字的同时却统一了彼此的计算机系统。在这种情况下，系统中难道还要再增加一条“purple”的特例吗？

软件架构师应该创建一个调度请求创建组件，并让该组件使用一个配置数据库来保存 URI 组装格式，这样的方式可以保护系统不受外界因素变化的影响。例如其配置信息可以如下：

URI	调度格式
Acme.com	/pickupAddress/%s/pickupTime/%s/dest/%s
.	/pickupAddress/%s/pickupTime/%s/destination/%s

但这样一来，软件架构师就需要通过增加一个复杂的组件来应对并不完全能实现互相替换的 restful 服务接口。

本章小结

LSP 可以且应该被应用于软件架构层面，因为一旦违背了可替换性，该系统架构就不得不为此增添大量复杂的应对机制。

第 10 章

ISP：接口隔离原则

“接口隔离原则（ISP）”这个名字来自图 10.1 所示的这种软件结构。

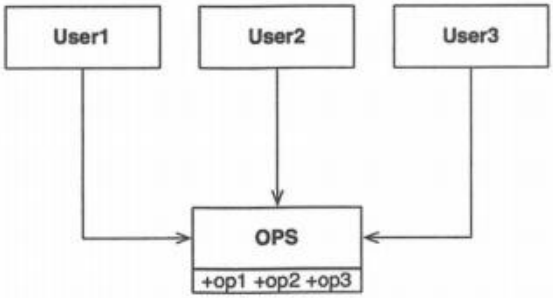


图 10.1：接口隔离原则

在图 10.1 所描绘的应用中，有多个用户需要操作 OPS 类。现在，我们假设这里的 User1 只需要使用 op1，User2 只需要使用 op2，User3 只需要使用 op3。

在这种情况下，如果 OPS 类是用 Java 编程语言编写的，那么很明显，User1 虽然不需要调用 op2、op3，但在源代码层次上也与它们形成依赖关系。这种依赖意味着我们对 OPS 代码中 op2 所做的任何修改，即使不会影响到 User1 的功能，也会导致它需要被重新编译和部署。

这个问题可以通过将不同的操作隔离成接口来解决，具体如图 10.2 所示。

同样，我们也假设这个例子是用 Java 这种静态类型语言来实现的，那么现在 User1 的源代码会依赖于 U1Ops 和 op1，但不会依赖于 OPS。这样一来，我们之后对 OPS 做的修改只要不影响到 User1 的功能，就不需要重新编译和部署 User1 了。

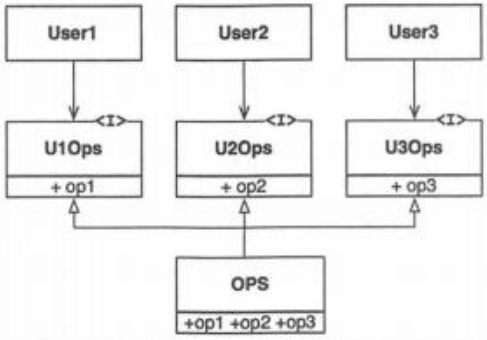


图 10.2：接口隔离

ISP 与编程语言

很明显，上述例子很大程度上也依赖于我们所采用的编程语言。对于 Java 这样的静态类型语言来说，它们需要程序员显式地 `import`、`use` 或者 `include` 其实现功能所需要的源代码。而正是这些语句带来了源代码之间的依赖关系，这也就导致了某些模块需要被重新编译和重新部署。

而对于 Ruby 和 Python 这样的动态类型语言来说，源代码中就不存在这样的声明，它们所用对象的类型会在运行时被推演出来，所以也就不存在强制重新编译和重新部署的必要性。这就是动态类型语言要比静态类型语言更灵活、耦合度更松的原因。

当然，如果仅仅就这样说的话，读者可能会误以为 ISP 只是一个与编程语言的选择紧密相关的设计原则，而非软件架构问题，这就错了。

ISP 与软件架构

回顾一下 ISP 最初的成因：在一般情况下，任何层次的软件设计如果依赖于不需要的东西，都会是有害的。从源代码层次来说，这样的依赖关系会导致不必要的重新编译和重新部署，对更高层次的软件架构设计来说，问题也是类似的。

例如，我们假设某位软件架构师在设计系统 S 时，想要在该系统中引入某个框架 F。这时候，假设框架 F 的作者又将其捆绑在一个特定的数据库 D 上，那么就形成了 S 依赖于 F，F 又依赖于 D 的关系（参见图 10.3）。

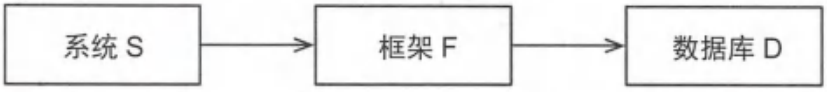


图 10.3：有问题的软件架构

在这种情况下，如果 D 中包含了 F 不需要的功能，那么这些功能同样也会是 S 不需要的。而我们对 D 中这些功能的修改将会导致 F 需要被重新部署，后者又会导

致 S 的重新部署。更糟糕的是，D 中一个无关功能的错误也可能会导致 F 和 S 运行出错。

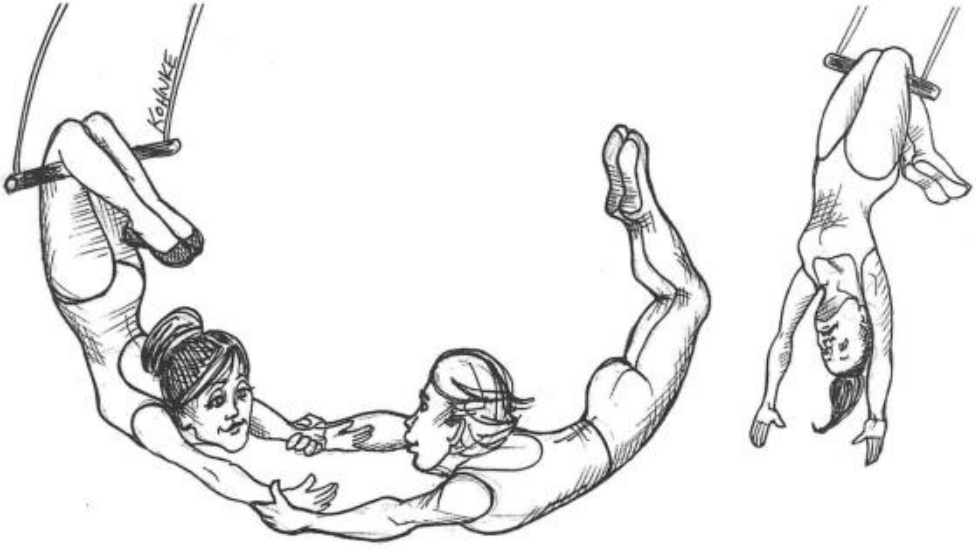
本章小结

本章所讨论的设计原则告诉我们：任何层次的软件设计如果依赖了它并不需要的东西，就会带来意料之外的麻烦。

我们将会在第 13 章“组件聚合”中讨论共同复用原则的时候再来深入探讨更多相关的细节。

第 11 章

DIP：依赖反转原则



依赖反转原则（DIP）主要想告诉我们的是，如果想要设计一个灵活的系统，在源代码层次的依赖关系中就应该多引用抽象类型，而非具体实现。

也就是说，在 Java 这类静态类型的编程语言中，在使用 use、import、include 这些语句时应该只引用那些包含接口、抽象类或者其他抽象类型声明的源文件，不应该引用任何具体实现。

同样的，在 Ruby、Python 这类动态类型的编程语言中，我们也不应该在源代码层次上引用包含具体实现的模块。当然，在这类语言中，事实上很难清晰界定某个模块是否属于“具体实现”。

显而易见，把这条设计原则当成金科玉律来加以严格执行是不现实的，因为软件系统在实际构造中不可避免地需要依赖到一些具体实现。例如，Java 中的 String 类就是这样一个具体实现，我们将其强迫转化为抽象类是不现实的，而在源代码层次上也无法避免对 java.lang.String 的依赖，并且也不应该尝试去避免。

但 String 类本身是非常稳定的，因为这个类被修改的情况是非常罕见的，而且可修改的内容也受到严格的控制，所以程序员和软件架构师完全不必担心 String 类上会发生经常性的或意料之外的修改。

同理，在应用 DIP 时，我们也不必考虑稳定的操作系统或者平台设施，因为这些系统接口很少会有变动。

我们主要应该关注的是软件系统内部那些会经常变动的（*volatile*）具体实现模块，这些模块是不停开发的，也就会经常出现变更。

稳定的抽象层

我们每次修改抽象接口的时候，一定也会去修改对应的具体实现。但反过来，当我们修改具体实现时，却很少需要去修改相应的抽象接口。所以我们可以认为接口比实现更稳定。

的确，优秀的软件设计师和架构师会花费很大精力来设计接口，以减少未来对

其进行改动。毕竟争取在不修改接口的情况下为软件增加新的功能是软件设计的基础常识。

也就是说，如果想要在软件架构设计上追求稳定，就必须多使用稳定的抽象接口，少依赖多变的具体实现。下面，我们将该设计原则归结为以下几条具体的编码守则：

- **应在代码中多使用抽象接口，尽量避免使用那些多变的具体实现类。**这条守则适用于所有编程语言，无论静态类型语言还是动态类型语言。同时，对象的创建过程也应该受到严格限制，对此，我们通常会选择用抽象工厂（*abstract factory*）这个设计模式。
- **不要在具体实现类上创建衍生类。**上一条守则虽然也隐含了这层意思，但它还是值得被单独拿出来做一次详细声明。在静态类型的编程语言中，继承关系是所有一切源代码依赖关系中最强的、最难被修改的，所以我们对继承的使用应该格外小心。即使是在稍微便于修改的动态类型语言中，这条守则也应该被认真考虑。
- **不要覆盖（override）包含具体实现的函数。**调用包含具体实现的函数通常就意味着引入了源代码级别的依赖。即使覆盖了这些函数，我们也无法消除这其中的依赖——这些函数继承了那些依赖关系。在这里，控制依赖关系的唯一办法，就是创建一个抽象函数，然后再为该函数提供多种具体实现。
- **应避免在代码中写入与任何具体实现相关的名字，或者是其他容易变动的事物的名字。**这基本上是 DIP 原则的另外一个表达方式。

工厂模式

如果想要遵守上述编码守则，我们就必须要对那些易变对象的创建过程做一些特殊处理，这样的谨慎是很有必要的，因为基本在所有的编程语言中，创建对象的操作都免不了需要在源代码层次上依赖对象的具体实现。

在大部分面向对象编程语言中，人们都会选择用抽象工厂模式来解决这个源代码依赖的问题。

下面，我们通过图 11.1 来描述一下该设计模式的结构。如你所见，Application 类是通过 Service 接口来使用 ConcreteImpl 类的。然而，Application 类还是必须要构造 ConcreteImpl 类实例。于是，为了避免在源代码层次上引入对 ConcreteImpl 类具体实现的依赖，我们现在让 Application 类去调用 ServiceFactory 接口的 makeSvc 方法。这个方法就由 ServiceFactoryImpl 类来具体提供，它是 ServiceFactory 的一个衍生类。该方法的具体实现就是初始化一个 ConcreteImpl 类的实例，并且将其以 Service 类型返回。

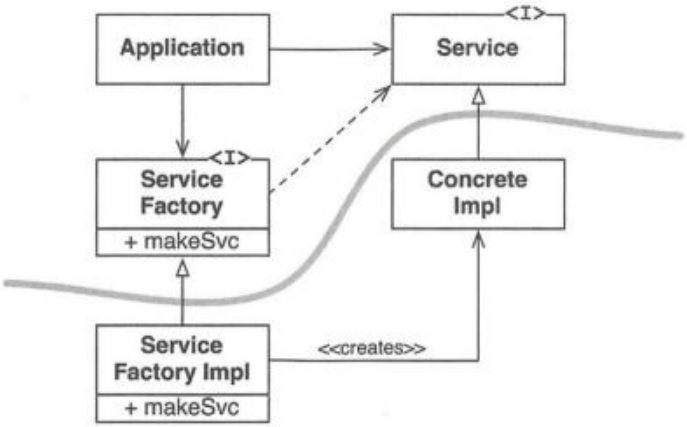


图 11.1：利用抽象工厂模式来管理依赖关系

图 11.1 中间的那条曲线代表了软件架构中的抽象层与具体实现层的边界。在这里，所有跨越这条边界源代码级别的依赖关系都应该是单向的，即具体实现层依赖抽象层。

这条曲线将整个系统划分为两部分组件：抽象接口与其具体实现。抽象接口组件中包含了应用的所有高阶业务规则，而具体实现组件中则包括了所有这些业务规则所需要做的具体操作及其相关的细节信息。

请注意，这里的控制流跨越架构边界的方向与源代码依赖关系跨越该边界的方向正好相反，源代码依赖方向永远是控制流方向的反转——这就是 DIP 被称为依赖反转原则的原因。

具体实现组件

在图 11.1 中，具体实现组件的内部仅有一条依赖关系，这条关系其实是违反 DIP 的。这种情况很常见，我们在软件系统中并不可能完全消除违反 DIP 的情况。通常只需要把它们集中于少部分的具体实现组件中，将其与系统的其他部分隔离即可。

绝大部分系统中都至少存在一个具体实现组件——我们一般称之为 `main` 组件，因为它们通常是 `main` 函数[1]所在之处。在图 11.1 中，`main` 函数应该负责创建 `ServiceFactoryImpl` 实例，并将其赋值给类型为 `ServiceFactory` 的全局变量，以便让 `Application` 类通过这个全局变量来进行相关调用。

本章小结

随着本书内容的进一步深入，以及我们对高级系统架构理论的进一步讨论，DIP 出现的频率将会越来越高。在系统架构图中，DIP 通常是最显而易见的组织原则。我们在后续章节中会把图 11.1 中的那条曲线称为架构边界，而跨越边界的、朝向抽象层的单向依赖关系则会成为一个设计守则——依赖守则。

1 这里指代的是操作系统用来启动应用程序的那个函数。

第 4 部分

组件构建原则

大型软件系统的构建过程与建筑物修建很类似，都是由一个个小组件组成的。所以，如果说 SOLID 原则是用于指导我们如何将砖块砌成墙与房间的，那么组件构建原则就是用来指导我们如何将这些房间组合成房子的。

在第 4 部分中，我们会详细讨论软件组件是什么，它们由什么元素构成，以及如何利用组件来构建系统。

第 12 章

组件

组件是软件的部署单元，是整个软件系统在部署过程中可以独立完成部署的最小实体。例如，对于 Java 来说，它的组件是 jar 文件。而在 Ruby 中，它们是 gem 文件。在.Net 中，它们则是 DLL 文件。总而言之，在编译运行语言中，组件是一组二进制文件的集合。而在解释运行语言中，组件则是一组源代码文件的集合。无论采用什么编程语言来开发软件，组件都是该软件在部署过程中的最小单元。

我们可以将多个组件链接成一个独立可执行文件，也可以将它们汇总成类似.war 文件这样的部署单元，又或者，组件也可以被打包成.jar、.dll 或者.exe 文件，并以可动态加载的插件形式来独立部署。但无论采用哪种部署形式，设计良好的组件都应该永远保持可被独立部署的特性，这同时也意味着这些组件应该可以被单独开发。

组件发展史

在早期的软件开发中，程序员可以完全掌控自己编写的程序所处的内存地址和存放格式。在那时，程序中的第一条语句被称为*起源（origin）语句*，它的作用是声明该程序应该被加载到的内存位置。

例如下面这段简单的 PDP-8 程序。该程序中包含一段名为 GETSTR 的子程序，作用是从键盘上读取一个字符串，并将其存入缓冲区。同时，该程序中还包含一段用于测试 GETSTR 功能的单元测试。

```
          *200
          TLS
START,    CLA
          TAD BUFR
          JMS GETSTR
          CLA
          TAD BUFR
          JMS PUTSTR
          JMP START
BUFR,     3000

GETSTR,   0
```

```
          DCA PTR
NXTCH,    KSF
          JMP -1
          KRB
          DCA I PTR
          TAD I PTR
          AND K177
          ISZ PTR
          TAD MCR
          SZA
          JMP NXTCH

K177,     177
MCR,      -15
```

首先，程序开头的*200 命令告诉编译器生成后的代码应该加载到内存地址为200（八进制）的位置。

当然，上面这种编程方式如今应该已经很少见了，因为现在的程序员一般不需要考虑程序要加载的内存地址。但这的确是早期程序员们在编程初期就要做的一个重要决策，因为当时的程序基本不能被重定位（relocate）。

那么，当时是如何调用库函数呢？上述代码演示了具体调用过程。程序员们需要将所有要调用的库函数源代码包含到自己的程序代码中，然后再进行整体编译[1]。库函数文件都是以源代码而非二进制的形式保存的。

在那个年代，存储设备十分缓慢，而内存则非常昂贵，也非常有限。编译器在编译程序的过程中需要数次遍历整个源代码。由于内存非常有限，驻留所有的源代码是不现实的，编译器只能多次从缓慢的存储设备中读取源代码。

这样做是十分耗时的——库函数越多，编译就越慢。大型程序的编译过程经常需要几个小时。

为了缩短编译时间，程序员们改将库函数的源代码单独编译。而库函数的源代

1 当年，我就职的第一家公司会将几十盒库函数源代码（打孔纸卡盒）按顺序摆放在一个书架上，当编写新程序时，只需要将需要用到的库源代码盒取下来，放在新编写代码的最后即可。

码在单独编译后会被加载到一个指定位置，比如地址 2000（八进制）。然后，编译器会针对该库文件创建一个符号表（symbol table），并将其和应用程序代码编译在一起。当程序运行时，它会先加载二进制形式的库文件[1]，再加载编译过的应用程序，其内存布局如图 12.1 所示。

图 12.1：早期程序的内存布局

当然，只要应用程序的代码能够完全存放在地址 0000~1777（八进制）内，这种组织方式就没有任何问题。但是，应用程序代码的大小很快就会超出这个范围。为了解决这个问题，程序员们必须将应用程序代码切分成两个不同的地址段，以跳过库函数存放的内存范围（具体如图 12.2 所示）。

图 12.2　将应用程序内存地址切分成两段

1　事实上，大部分的古典机型采用的都是 CORE 内存，后者内存中的内容在关机时不会丢失，库函数经常会留在内存中数日之久。

很显然，这种方案也是不可持续的。因为随着函数库中函数的增加，它的大小也随之增加，我们同样也需要为此划分新的区域，譬如在上述例子中，我们需要在7000（八进制）左右的位置往后追加地址空间。这样一来，程序和函数库的碎片化程度会随着计算机内存的增加而不断增加。

显而易见，这个问题必须要有一个解决方案。

重定位技术

该解决方案就是生成可重定位的二进制文件。其背后的原理非常简单，即程序员修改编译器输出文件的二进制格式，使其可以由一个智能加载器加载到任意内存位置。当然，这需要我们在加载器启动时为这些文件指定要加载到的内存地址，而且可重定位的代码中还包含了一些记号，加载器将其加载到指定位置时会修改这些记号对应的地址值。一般来说，这个过程只不过就是将二进制文件中包含的内存地址都按照其加载到的内存基础位置进行递增。

这样一来，程序员们就可以用加载器来调整函数库及应用程序的位置了。事实上，这种加载器还可以接受多个二进制文件的输入，并按顺序在内存中加载它们，再逐个进行重定位。这样，程序员们就可以只加载他们实际会用到的函数了。

除此之外，程序员们还对编译器做了另外一个修改，就是在可重定位二进制文件中将函数名输出为元数据并存储起来。这样一来，如果一段程序调用了某个库函数，编译器就会将这个函数的名字输出为外部引用（*external reference*），而将库函数的定义输出为外部定义（*external definition*）。加载器在加载完程序后，会将外部引用和外部定义链接（*link*）起来。

这就是链接加载器（linking loader）的由来。

链接器

链接加载器让程序员们可以将程序切分成多个可被分别编译、加载的程序段。

在程序规模较小、外部链接也较少的情况下，这个方案一直都很好用。然而在20世纪60年代末期到70年代初期的那段时间里，程序的规模突然有了大幅的增长，情况就有所不同了。

显然在这种情况下，链接加载器的处理过程实在是太慢了。且不说函数库当时还存储在磁带卷这样的低速存储设备上，即使是存储在磁盘上，其存取速度也是很慢的。毕竟，链接加载器在加载处理过程中必须要读取几十个甚至几百个二进制库文件来解析外部引用。因此随着程序规模的扩大，以及函数库中函数的累积，链接加载器的加载过程经常会出现需要一个多小时才能完成的情况。

最后，程序员们只能将加载过程和链接过程也进行分离。他们将耗时较长的部分——链接部分——放到了一个单独的程序中去进行，这个程序就是所谓的链接器（*linker*）。链接器的输出是一个已经完成了外部链接的、可以重定位的二进制文件，这种文件可以由一个支持重定位的加载器迅速加载到内存中。这使得程序员可以用缓慢的链接器生产出可以很快进行多次加载的可执行文件。

时间继续推移到了20世纪80年代，程序员们在那时已经用上了C这样的高级编程语言，程序的规模也得到了进一步的扩大，源代码行数超过几十万行在当时已经是很普遍的事了。

于是，源代码模块会从.c文件被编译成.o文件，然后再由链接器创建出可被快速加载的可执行文件。那时，虽然编译每个单独模块的速度相对较快，但所有模块的累计编译时间较长，链接过程则耗时更久，整个修改编译周期经常会超过数个小时。

有时候，程序员们看上去似乎就是一直不停地在原地打转。从20世纪60年代一直到80年代，他们所有为提供编译速度所做的努力都被不断增长的程序规模抵消了。程序员好像永远也脱离不了长达几个小时的修改编译周期。程序加载的速度一直都很快，但是其编译和链接的过程也一直是整个开发过程的瓶颈。

这被我们称为程序规模上的墨菲定律：

> 程序的规模会一直不断地增长下去，直到将有限的编译和链接时间填满为止。

除了墨菲定律，我们还有摩尔定律[1]。在 20 世纪 80 年代，两个定律一直在互相较量，最终以摩尔定律获胜告终。因为磁盘的物理尺寸一直在不断缩小，速度在不断提高，同时内存的造价也一直在不断降低，以至于大部分存放在磁盘上的数据都可以被缓存在内存中了。而计算机时钟频率则从 1MHz 上升到了 100MHz。

到了 20 世纪 90 年代中期，链接速度的提升速度已经远远超过了程序规模的增长速度。在大部分情况下，程序链接的时间已经降低到秒级。这对一些小程序来说，即使使用链接加载器也是可以接受的了。

与此同时，编程领域中还诞生了 Active-X、共享库、.jar 文件等组件形式。由于计算与存储速度的大幅提高，我们又可以在加载过程中进行实时链接了，链接几个.jar 文件或是共享库文件通常只需要几秒钟时间，由此，插件化架构也就随之诞生了。

如今，我们用.jar 文件、DLL 文件和共享库方式来部署应用的插件已经非常司空见惯了。如果现在我们想要给 Minecraft 增加一个模块，只需要将.jar 文件放到一个指定的目录中即可。同样的，如果你想给 Visual Studio 增加 Resharper 插件，也只需要安装对应的 DLL 文件即可。

本章小结

我们常常会在程序运行时插入某些动态链接文件，这些动态链接文件所使用的就是软件架构中的组件概念。在经历了 50 年的演进之后，组件化的插件式架构已经成为我们习以为常的软件构建形式了。

1 摩尔定律：计算机的处理速度、内存、存储密度每 18 个月会增长 1 倍。这条定律从 1950 年到 2000 年一直适用，之后在处理速度方面就停滞不前了。

第 13 章

组件聚合

那么，究竟是哪些类应该被组合成一个组件呢？这是一个非常重要的设计决策，应该遵循优秀的软件工程经验来行事。但不幸的是，很多年以来，我们对于这么重要的决策经常是根据当下面临的实际情况临时拍脑门决定的。

在本章中，我们会具体讨论以下三个与构建组件相关的基本原则：

- REP：复用/发布等同原则。
- CCP：共同闭包原则。
- CRP：共同复用原则。

复用/发布等同原则

> 软件复用的最小粒度应等同于其发布的最小粒度。

过去十年间，模块管理工具得到了长足的发展，例如 Maven、Leiningen、RVM 等。这些工具日益重要的原因是正好在这十年间出现了大量可复用的组件和组件库。应该说，我们现在至少已经实现了面向对象编程的一个原始初衷——软件复用。

REP 原则初看起来好像是不言自明的。毕竟如果想要复用某个软件组件的话，一般就必须要求该组件的开发由某种发布流程来驱动，并且有明确的发布版本号。

这其中的一个原因是，如果没有设定版本号，我们就没有办法保证所有被复用的组件之间能够彼此兼容。另外更重要的一点是，软件开发者必须要能够知道这些组件的发布时间，以及每次发布带来了哪些变更。

只有这样，软件工程师才能在收到相关组件新版本发布的通知之后，依据该发布所变更的内容来决定是继续使用旧版本还是做些相应的升级，这是很基本的要求。因此，组件的发布过程还必须要能够产生适当的通知和发布文档，以便让它的用户根据这些信息做出有效的升级决策。

从软件设计和架构设计的角度来看，REP 原则就是指组件中的类与模块必须是彼此紧密相关的。也就是说，一个组件不能由一组毫无关联的类和模块组成，它们之间应该有一个共同的主题或者大方向。

但从另外一个视角来看，这个原则就没那么简单了。因为根据该原则，一个组件中包含的类与模块还应该是可以同时发布的。这意味着它们共享相同的版本号与版本跟踪，并且包含在相同的发行文档中，这些都应该同时对该组件的作者和用户有意义。

这层建议听起来就比较薄弱了，毕竟说某项事情的安排应该“合理”的确有点假大空，不着实际。该建议薄弱的原因是它没有清晰地定义出到底应该如何将类与模块组合成组件。但即使这样，REP 原则的重要性也是毋庸置疑的，因为违反这个原则的后果事实上很明显——一定会有人抱怨你的安排“不合理”，并进而对你的软件架构能力产生怀疑。

而且，REP 原则的上述薄弱性也会由以下两个原则所补充。CCP 和 CRP 会从相反的角度对这个原则进行有力的补偿。

共同闭包原则

> 我们应该将那些会同时修改，并且为相同目的而修改的类放到同一个组件中，而将不会同时修改，并且不会为了相同目的而修改的那些类放到不同的组件中。

这其实是 SRP 原则在组件层面上的再度阐述。正如 SRP 原则中提到的“一个类不应该同时存在着多个变更原因”一样，CCP 原则也认为一个组件不应该同时存在着多个变更原因。

对大部分应用程序来说，可维护性的重要性要远远高于可复用性。如果某程序中的代码必须要进行某些变更，那么这些变更最好都体现在同一个组件中，而不是分布于很多个组件中[1]。因为如果这些变更都集中在同一个组件中，我们就只需要重新部署该组件，其他组件则不需要被重新验证、重新部署了。

总而言之，CCP 的主要作用就是提示我们要将所有可能会被一起修改的类集中

1　请参阅第 27 章“服务：宏观与微观”中的“运送猫咪的难题”。

在一处。也就是说，如果两个类紧密相关，不管是源代码层面还是抽象理念层面，永远都会一起被修改，那么它们就应该被归属为同一个组件。通过遵守这个原则，我们就可以有效地降低因软件发布、验证及部署所带来的工作压力。

另外，CCP 原则和开闭原则（OCP）也是紧密相关的。CCP 讨论的就是 OCP 中所指的“闭包”。OCP 原则认为一个类应该便于扩展，而抗拒修改。由于 100% 的闭包是不可能的，所以我们只能战略性地选择闭包范围。在设计类的时候，我们需要根据历史经验和预测能力，尽可能地将需要被一同变更的那些点聚合在一起。

对于 CCP，我们还可以在此基础上做进一步的延伸，即可以将某一类变更所涉及的所有类尽量聚合在一处。这样当此类变更出现时，我们就可以最大限度地做到使该类变更只影响到有限的相关组件。

与 SRP 原则的相似点

如前所述，CCP 原则实际上就是 SRP 原则的组件版。在 SRP 原则的指导下，我们将会把变更原因不同的函数放入不同的类中。而 CCP 原则指导我们应该将变更原因不同的类放入不同的组件中。简而言之，这两个原则都可以用以下一句简短的话来概括：

> 将由于相同原因而修改，并且需要同时修改的东西放在一起。将由于不同原因而修改，并且不同时修改的东西分开。

共同复用原则

> 不要强迫一个组件的用户依赖他们不需要的东西。

共同复用原则（CRP）是另外一个帮助我们决策类和模块归属于哪一个组件的原则。该原则建议我们将经常共同复用的类和模块放在同一个组件中。

通常情况下，类很少会被单独复用。更常见的情况是多个类同时作为某个可复用的抽象定义被共同复用。CRP 原则指导我们将这些类放在同一个组件中，而在这样的组件中，我们应该预见到会存在着许多互相依赖的类。

一个简单的例子就是容器类与其相关的遍历器类，这些类之间通常是紧密相关的，一般会被共同复用，因此应该被放置在同一个组件中。

但是 CRP 的作用不仅是告诉我们应该将哪些类放在一起，更重要的是要告诉我们应该将哪些类分开。因为每当一个组件引用了另一个组件时，就等于增加了一条依赖关系。虽然这个引用关系仅涉及被引用组件中的一个类，但它所带来的依赖关系丝毫没有减弱。也就是说，引用组件已然依赖于被引用组件了。

由于这种依赖关系的存在，每当被引用组件发生变更时，引用它的组件一般也需要做出相应的变更。即使它们不需要进行代码级的变更，一般也免不了需要被重新编译、验证和部署。哪怕引用组件根本不关心被引用组件中的变更，也要如此。

因此，当我们决定要依赖某个组件时，最好是实际需要依赖该组件中的每个类。换句话说，我们希望组件中的所有类是不能拆分的，即不应该出现别人只需要依赖它的某几个类而不需要其他类的情况。否则，我们后续就会浪费不少时间与精力来做不必要的组件部署。

因此在CRP原则中，关于哪些类不应该被放在一起的建议是其更为重要的内容。简而言之，CRP 原则实际上是在指导我们：不是紧密相连的类不应该被放在同一个组件里。

与 ISP 原则的关系

CRP 原则实际上是 ISP 原则的一个普适版。ISP 原则是建议我们不要依赖带有不需要的函数的类，而 CRP 原则则是建议我们不要依赖带有不需要的类的组件。

上述两条建议实际上都可以用下面一句话来概括：

> 不要依赖不需要用到的东西。

组件聚合张力图

读到这里，读者可能已经意识到上述三个原则之间彼此存在着竞争关系。REP

和 CCP 原则是黏合性原则，它们会让组件变得更大，而 CRP 原则是排除性原则，它会尽量让组件变小。软件架构师的任务就是要在这三个原则中间进行取舍。

下面我们来看一下图 13.1。这是一张组件聚合三大原则的张力图[1]，图的边线所描述的是忽视对应原则的后果。

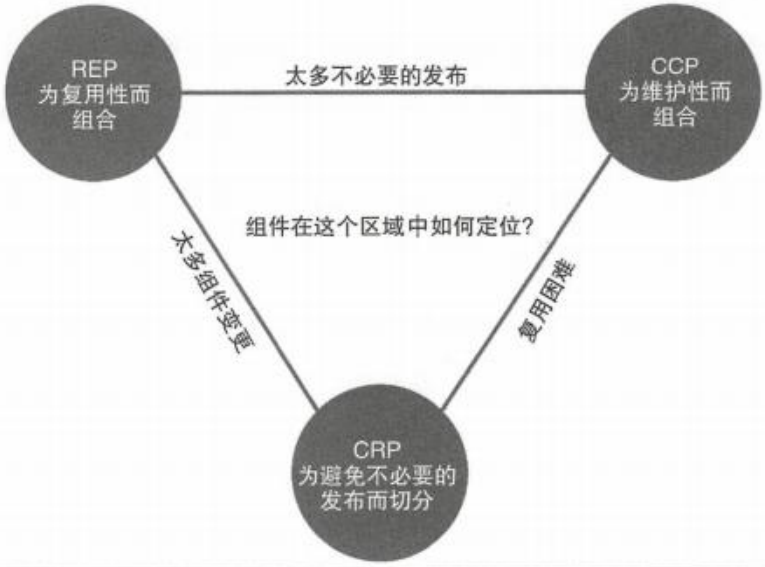


图 13.1：组件聚合原则的张力图

简而言之，只关注 REP 和 CRP 的软件架构师会发现，即使是简单的变更也会同时影响到许多组件。相反，如果软件架构师过于关注 CCP 和 REP，则会导致很多不必要的发布。

优秀的软件架构师应该能在上述三角张力区域中定位一个最适合目前研发团队状态的位置，同时也会根据时间不停调整。例如在项目早期，CCP 原则会比 REP 原则更重要，因为在这一阶段研发速度比复用性更重要。

一般来说，一个软件项目的重心会从该三角区域的右侧开始，先期主要牺牲的是复用性。然后，随着项目逐渐成熟，其他项目会逐渐开始对其产生依赖，项目重

1 感谢 Tom Ottinger 最先提出这个想法。

心就会逐渐向该三角区域的左侧滑动。换句话说，一个项目在组件结构设计上的重心是根据该项目的开发时间和成熟度不断变动的，我们对组件结构的安排主要与项目开发的进度和它被使用的方式有关，与项目本身功能的关系其实很小。

本章小结

过去，我们对组件在构建过程中要遵循的组合原则的理解要比 REP、CCP、CRP 这三个原则更有限。我们最初所理解的组合原则可能完全基于单一职责原则。然而，本章介绍的这三个原则为我们描述了一个更为复杂的决策过程。在决定将哪些类归为同一个组件时，必须要考虑到研发性与复用性之间的矛盾，并根据应用程序的需要来平衡这两个矛盾，这是一件很不容易的事。而且，这种平衡本身也在不断变化。也就是说，当下适用的分割方式可能明年就不再适用了。所以，组件的构成安排应随着项目重心的不同，以及研发性与复用性的不同而不断演化。

第 14 章

组件耦合

接下来要讨论的三条原则主要关注的是组件之间的关系。在这些原则中，我们同样会面临着研发能力和逻辑设计之间的冲突。毕竟，影响组件结构的不仅有技术水平和公司内部政治斗争这两个因素，其结构本身更是不断变化的。

无依赖环原则

组件依赖关系图中不应该出现环。

我们一定都有过这样的经历：当你花了一整天的时间，好不容易搞定了一段代码，第二天上班时却发现这段代码莫名其妙地又不能工作了。这通常是因为有人在你走后修改了你所依赖的某个组件。我给这种情况起了个名字——“一觉醒来综合征”。

这种综合征的主要病因是多个程序员同时修改了同一个源代码文件。虽然在规模相对较小、人员较少的项目中，这种问题或许并不严重，但是随着项目的增长，研发人员的增加，这种每天早上刚上班时都要经历一遍的痛苦就会越来越多。甚至会严重到让有的团队在长达数周的时间内都不能发布一个稳定的项目版本，因为每个人都在不停地修改自己的代码，以适应其他人所提交的变更。

在过去几十年中，针对这个问题逐渐演化出了两种解决方案，它们都来自电信行业。第一种是“每周构建”，第二种是“无依赖环原则（ADP）”。

每周构建

每周构建方案是中型项目中很常见的一种管理手段。其具体做法如下：在每周的前四天中，让所有的程序员在自己的私有库上工作，忽略其他人的修改，也不考虑互相之间的集成问题；然后在每周五要求所有人将自己所做的变更提交，进行统一构建。

上述方案确实可以让程序员们每周都有四天的时间放手干活。然而一到星期五，所有人都必须要花费大量的精力来处理前四天留下来的问题。

而且更不幸的是，随着项目越来越大，每周五的集成工作会越来越难以按时完成。而随着集成任务越来越重，周六的加班也会变得越来越频繁。经历过几次这样的加班之后，就会有人提出应该将集成任务提前到星期四开始，就这样一步一步地，集成工作慢慢地就要占用掉差不多半周的时间。

事实上，这个问题最终还会造成更大的麻烦。因为如果我们想要保持高效率的开发，就不能频繁地进行构建操作，但是如果我们减少了构建次数，延长了项目被构建的时间间隔，又会影响到该项目的质量，增大它的风险。整个项目会变得越来越难以构建与测试，团队反馈周期会越来越长，研发质量自然也会越来越差。

消除循环依赖

对于上述情景，我们的解决办法是将研发项目划分为一些可单独发布的组件，这些组件可以交由单人或者某一组程序员来独立完成。当有人或团队完成某个组件的某个版本时，他们就会通过发布机制通知其他程序员，并给该组件打一个版本号，放入一个共享目录。这样一来，每个人都可以依赖于这些组件公开发布的版本来进行开发，而组件开发者则可以继续去修改自己的私有版本。

每当一个组件发布新版本时，其他依赖这个组件的团队都可以自主决定是否立即采用新版本。若不采用，该团队可以选择继续使用旧版组件，直到他们准备好采用新版本为止。

这样就不会出现团队之间相互依赖的情况了。任何一个组件上的变更都不会立刻影响到其他团队。每个团队都可以自主决定是否立即集成自己所依赖组件的新版本。更重要的是，这种方法使我们的集成工作能以一种小型渐进的方式来进行。程序员们再也不需要集中在一起，统一集成相互的变更了。

如你所见，上述整个过程既简单又很符合逻辑，因而得到了各个研发团队的广泛采用。但是，如果想要成功推广这个开发流程，就必须控制好组件之间的依赖结构，*绝对不能允许该结构中存在着循环依赖关系*。如果某项目结构中存在着循环依赖关系，那么“一觉醒来综合征”将是不可避免的。

下面让我们来看看图 14.1，该图展示了一个典型应用程序的组件结构。当然，这个应用程序的具体功能与我们要讨论的无关，真正重要的是其组件之间的依赖结构。我们应该可以注意到，该组件依赖结构所呈现的是一个有向图，图中的每个节点都是一个项目组件，依赖关系就是有向图中的边。

更重要的是，不管我们从该图中的哪个节点开始，都不能沿着这些代表了依赖关系的边最终走回到起始点。也就是说，这种结构中不存在环，我们称这种结构为有向无环图（Directed Acyclic Graph，简写为 DAG）。

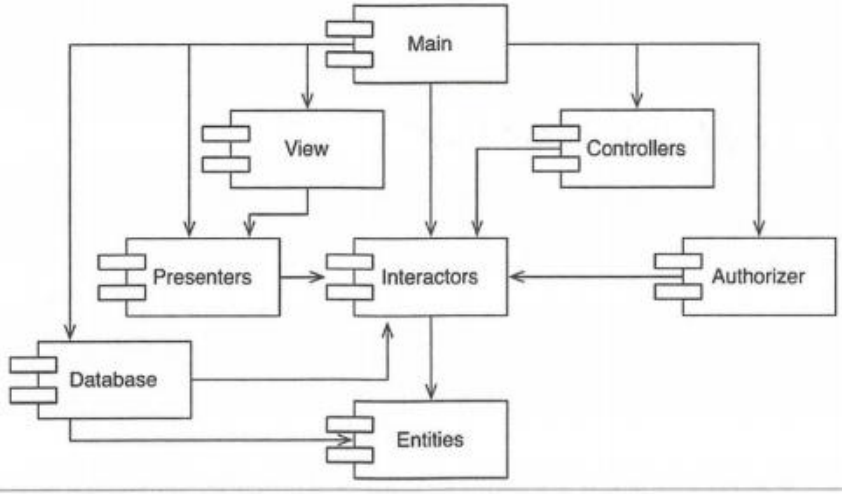


图 14.1：典型的组件结构图

现在，如果负责 Presenters 组件的团队需要发布一个新版本，我们就应该很容易判断出哪些组件会受这个变更的影响——只需要按其依赖关系反向追溯即可。显然在图 14.1 中，View 和 Main 是同时会被影响的两个组件。这两个组件的研发团队需要决定是否采用 Presenters 组件的新版本。

另外值得注意的是，当 Main 组件发布新版本时，它对系统中的其他组件根本就没有影响，既没有一个组件依赖于 Main，也就没有人需要关心 Main 组件上发生的变更。这太好了，至少表示我们在发布 Main 的新版本时，可以不必考虑它对整个项目微乎其微的影响。

当 Presenters 组件的程序员们需要进行一次测试时，他们只需要将对应版本的 Presenters 和 Interactors 及 Entities 的当前版本一起构建并测试即可，其他组件不需要做任何修改。这可以让编写 Presenters 组件的程序员们在编写测试时考虑更少的变量，工作量更小。

当我们需要发布整个系统时，可以让整个过程从下至上来进行。具体来说就是，首先对 Entities 组件进行编译、测试、发布。随后是 Database 和 Interactors 这两个组件。再紧随其后的是 Presenters、View、Controllers，以及 Authorizer 四个组件。最后是 Main 组件。这样一来，整个流程会非常清晰，也很容易。只要我们了解系统各部分之间的依赖关系，构建整套系统就会变得很容易。

循环依赖在组件依赖图中的影响

假设某个新需求使我们修改了 Entities 组件中的某个类，而这个类又依赖于 Authorizer 组件中的某个类。例如，Entities 组件中的 User 类使用了 Authorizer 组件中的 Permissions 类。这就形成了一个循环依赖关系，如图 14.2 所示。

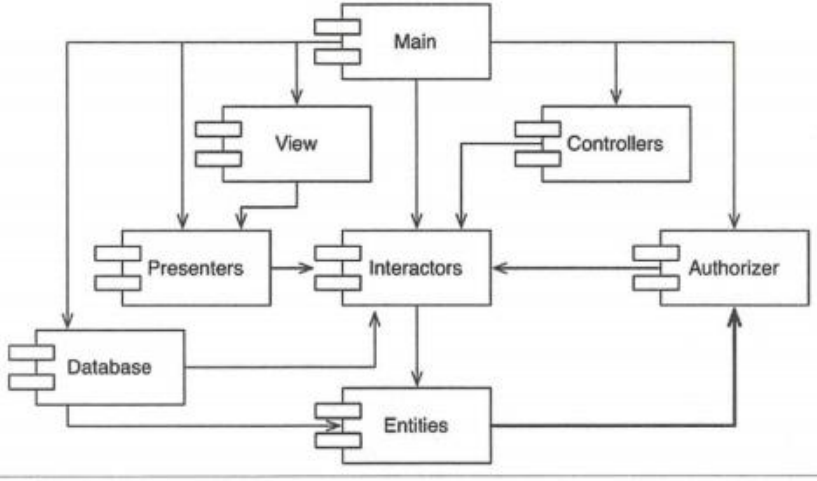


图 14.2：循环依赖

这种循环依赖立刻就会给我们的项目带来麻烦。例如，当 Database 组件的程序员需要发布新版本时，他们需要与 Entities 组件进行集成。但现在由于出现了循环依赖，Database 组件就必须也要与 Authorizer 组件兼容，而 Authorizer 组件又依赖于 Interactors 组件。这样一来，Database 组件的发布就会变得非常困难。在这里，Entities、Authorizer 及 Interactors 这三个组件事实上被合并成了一个更大的组件。这些组件的程序员现在会互相形成干扰，因为他们在开发中都必须使用完全相同的组件版本。

这还只是问题的冰山一角，请想象一下我们在测试 Entities 组件时会发生什么？情况会让人触目惊心，我们会发现自己必须将 Authorizer 和 Interactors 集成到一起测试。即使这不是不能容忍的事，但至少这些组件之间的耦合度也是非常令人不安的。

很显然，这样一个小小的测试必须要依赖大量的库就是因为其组件结构依赖图中存在的这个循环依赖。这种循环依赖会使得组件的独立维护工作变得十分困难。不仅如此，单元测试和发布流程也都会变得非常困难，并且很容易出错。此外，项目在构建中出现的问题会随着组件数量的增多而呈现出几何级数的增长。

所以，当组件结构依赖图中存在循环依赖时，想要按正确的顺序构建组件几乎是不可能的。这种依赖关系将会在 Java 这种需要在编译好的二级制文件中读取声明信息的语言中导致一些非常棘手的问题。

打破循环依赖

当然，我们可以打破这些组件中的循环依赖，并将其依赖图转化为 DAG。目前有以下两种主要机制可以做到这件事情。

1. 应用依赖反转原则（DIP）：在图 14.3 中，我们可以创建一个 User 类需要使用的接口，然后将这个接口放入 Entities 组件，并在 Authorizer 组件中继承它。这样就将 Entities 与 Authorizer 之间的依赖关系反转了，自然也就打破了循环依赖关系。

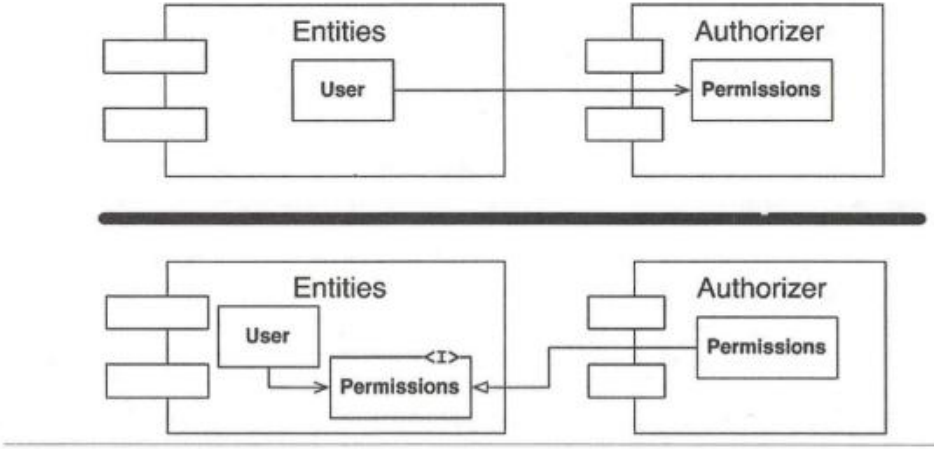


图 14.3：将 Entities 与 Authorizer 之间的依赖关系反转

2. 创建一个新的组件，并让 Entities 与 Authorize 这两个组件都依赖于它。将现有的这两个组件中互相依赖的类全部放入新组件（如图 14.4 所示）。

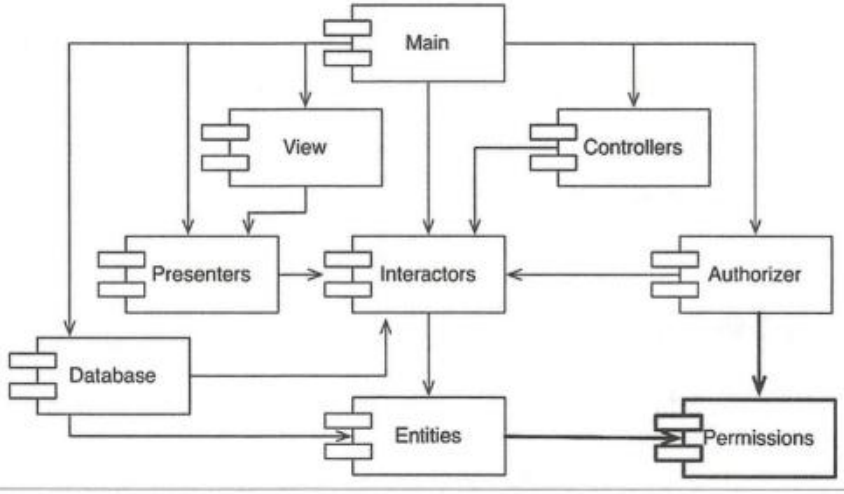


图 14.4：让 Entities 与 Authorizer 共同依赖于一个新组件

“抖动”

当然，采用第二种解决方案也意味着在需求变更时，项目的组件结构也要随之变更。确实是这样的，随着应用程序的不断演进，其组件结构也会不停地抖动和扩

张。因此，我们必须要持续地监控项目中的循环依赖关系。当循环依赖出现时，必须以某种方式消除它们。为此，我们有时候不可避免地需要创建新的组件，而使整个组件结构变得更大。

自上而下的设计

根据上述讨论，我们可以得出一个无法逃避的结论：组件结构图是不可能自上而下被设计出来的。它必须随着软件系统的变化而变化和扩张，而不可能在系统构建的最初就被完美设计出来。

有些读者可能会觉得这个结论有些反直觉。人们通常会直观地认为，代表项目粗粒度的结构单元，也就是组件，应该与顶层设计中的功能单元是相对应的。

同样的，人们也普遍认为项目粗粒度的组件分组规则所产生的就是组件的依赖结构，也应该在某种程度上与项目的系统功能分解的结果相互对应。但是很明显，组件依赖关系图其实不具备这样的属性。

事实上，组件依赖结构图并不是用来描述应用程序功能的，它更像是应用程序在构建性与维护性方面的一张地图。这就是组件的依赖结构图不能在项目的开始阶段被设计出来的原因——当时该项目还没有任何被构建和维护的需要，自然也就不需要一张地图来指引。然而，随着早期被设计并实现出来的模块越来越多，项目中就逐渐出现了要对组件依赖关系进行管理的需求，以此来预防“一觉醒来综合征”的爆发。除此之外，我们还希望将项目变更所影响的范围被限制得越小越好，因此需要应用单一职责原则（SRP）和共同闭包原则（CCP）来将经常同时被变更的类聚合在一起。

组件结构图中的一个重要目标是指导如何隔离频繁的变更。我们不希望那些频繁变更的组件影响到其他本来应该很稳定的组件，例如，我们通常不会希望无关紧要的 GUI 变更影响到业务逻辑组件；我们也不希望对报表的增删操作影响到其高阶策略。出于这样的考虑，软件架构师们才有必要设计并且铸造出一套组件依赖关系图来，以便将稳定的高价值组件与常变的组件隔离开，从而起到保护作用。

另外，随着应用程序的增长，创建可重用组件的需要也会逐渐重要起来。这时CRP又会开始影响组件的组成。最后当循环依赖出现时，随着无循环依赖原则（ADP）的应用，组件依赖关系会产生相应的抖动和扩张。

如果我们在设计具体类之前就来设计组件依赖关系，那么几乎是必然要失败的。因为在当下，我们对项目中的共同闭包一无所知，也不可能知道哪些组件可以复用，这样几乎一定会创造出循环依赖的组件。因此，组件依赖关系是必须要随着项目的逻辑设计一起扩张和演进的。

稳定依赖原则

依赖关系必须要指向更稳定的方向。

设计这件事不可能是完全静止的，如果我们要让一个设计是可维护的，那么其中某些部分就必须是可变的。通过遵守共同闭包原则（CCP），我们可以创造出对某些变更敏感，对其他变更不敏感的组件。这其中的一些组件在设计上就已经是考虑了易变性，预期它们会经常发生变更的。

任何一个我们预期会经常变更的组件都不应该被一个难于修改的组件所依赖，否则这个多变的组件也将会变得非常难以被修改。

这就是软件开发的困难之处，我们精心设计的一个容易被修改的组件很可能会由于别人的一条简单依赖而变得非常难以被修改。即使该模块中没有一行代码需要被修改，但是整个模块在被修改时所面临的挑战性也已经存在了。而通过遵守稳定依赖原则（SDP），我们就可以确保自己设计中那些容易变更的模块不会被那些难于修改的组件所依赖。

稳定性

我们该如何定义“稳定性”呢？譬如说将一个硬币立起来放，你认为它会处于一个稳定的位置吗？当然不会。然而，除非受到外界因素干扰，否则硬币本身可以在这个位置保持相当长的一段时间。因此稳定性应该与变更的频繁度没有直接关系。

但问题是硬币并没有倒，为什么我们却并不认为它是稳定的呢？

下面来看看 Webster 在线字典中的描述：稳定指的是“很难移动”。所以稳定性应该与变更所需的工作量有关。例如，硬币是不稳定的，因为只需要很小的动作就可以推倒它，而桌子则是非常稳定的，因为将它掀翻需要很大的动作。

但如果将这套理论关联到软件开发的问题上呢？软件组件的变更困难度与很多因素有关，例如代码的体量大小、复杂度、清晰度等。我们在这里会忽略这些因素，只集中讨论一个特别的因素——让软件组件难于修改的一个最直接的办法就是让很多其他组件依赖于它。带有许多入向依赖关系的组件是非常稳定的，因为它的任何变更都需要应用到所有依赖它的组件上。

在图 14.5 中，X 是一个稳定的组件。因为有三个组件依赖着 X，所以 X 有三个不应该被修改的原因。这里就说 X 要对三个组件负责。另一方面，X 不依赖于任何组件，所以不会有任何原因导致它需要被变更，我们称它为“独立”组件。

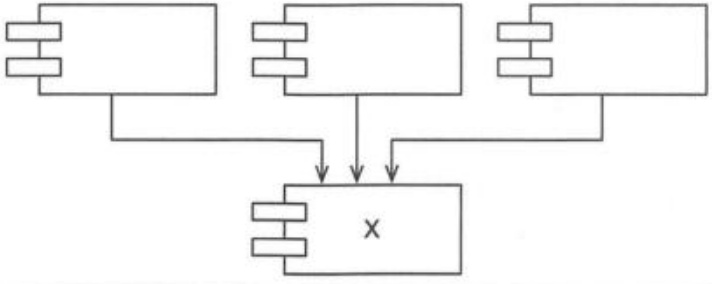


图 14.5：X，稳定的组件

下面再来看看图 14.6 中的 Y 组件，这是一个非常不稳定的组件。由于没有其他的组件依赖 Y，所以 Y 并不对任何组件负责。但因为 Y 同时依赖于三个组件，所以它的变更就可能由三个不同的源产生。这里就说 Y 是有依赖性的组件。

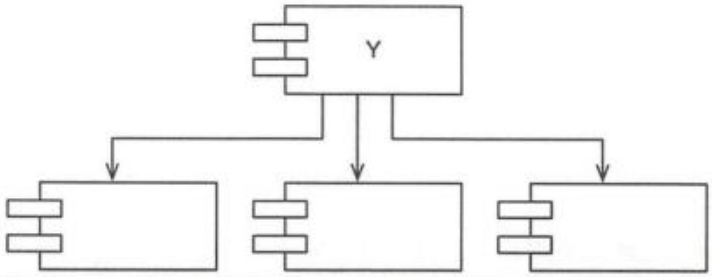


图 14.6：Y，一个非常不稳定的组件

稳定性指标

那么，究竟该如何来量化一个组件的稳定性呢？其中一种方法是计算所有入和出的依赖关系。通过这种方法，我们就可以计算出一个组件的位置稳定性（positional stability）。

- *Fan-in*：入向依赖，这个指标指代了组件外部类依赖于组件内部类的数量。
- *Fan-out*：出向依赖，这个指标指代了组件内部类依赖于组件外部类的数量。
- *I*：不稳定性，*I* = *Fan-out* / (*Fan-in* + *Fan-out*)。该指标的范围是[0,1]，*I*=0 意味着组件是最稳定的，*I*=1 意味着组件是最不稳定的。

在这里，*Fan-in* 和 *Fan-out* 这两个指标[1]是通过统计和组件内部类有依赖的组件外部类的数量来计算的，具体如图 14.7 所示。

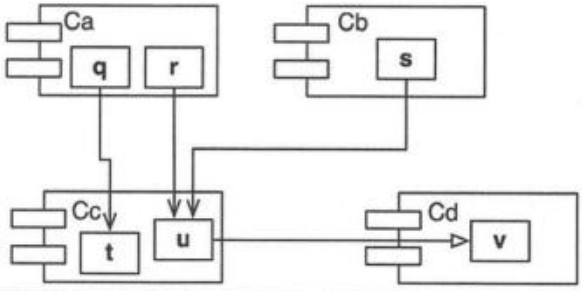


图 14.7：我们的例子

1 在之前的文章中，我曾经用过 *efferent*、*afferent*（简称 Ce 和 Ca）这两个术语来指代 *Fan-out* 和 *Fan-in*。那只是我的一个个人偏好：我喜欢用人的中枢神经系统来比喻软件架构。

在这里，我们想要计算组件 Cc 的稳定性指标，可以观察到有 3 个类在 Cc 外部，它们都依赖于 Cc 内部的类，因此 *Fan-in*=3。此外，Cc 中的一个类也依赖于组件外部的类，因此 *Fan-out*=1，*I*=1/4。

在 C++中，这些依赖关系一般是通过`#include`语句来表达的。事实上，当每个源文件只包含一个类的时候，*I* 指标是最容易计算的。同样在 Java 中，*I* 指标也可以通过 `Import` 语句和全引用名字的数量来计算。

当 *I* 指标等于 1 时，说明没有组件依赖当前组件（*Fan-in*=0），同时该组件却依赖于其他组件（*Fan-out*>0）。这是组件最不稳定的一种情况，我们认为这种组件是“不负责的（irresponsible）、对外依赖的（dependent）”。由于这个组件没有被其他组件依赖，所以自然也就没有力量会干预它的变更，同时也因为该组件依赖于其他组件，所以就必然会经常需要变更。

相反，当 *I*=0 的时候，说明当前组件是其他组件所依赖的目标（*Fan-in*>0），同时其自身并不依赖任何其他组件（*Fan-out*=0）。我们通常认为这样的组件是“负责的（*responsibile*）、不对外依赖的（*independent*）”。这是组件最具稳定性的一种情况，其他组件对它的依赖关系会导致这个组件很难被变更，同时由于它没有对外依赖关系，所以不会有来自外部的变更理由。

稳定依赖原则（SDP）的要求是让每个组件的 *I* 指标都必须大于其所依赖组件的 *I* 指标。也就是说，组件结构依赖图中各组件的 *I* 指标必须要按其依赖关系方向递减。

并不是所有组件都应该是稳定的

如果一个系统中的所有组件都处于最高稳定性状态，那么系统就一定无法再进行变更了，这显然不是我们想要的。事实上，我们设计组件架构图的目的就是要决定应该让哪些组件稳定，让哪些组件不稳定。譬如在图 14.8 中，我们所示范的就是一个具有三个组件的系统的理想配置。

在该系统组件结构图中，可变更的组件位于顶层，同时依赖于底层的稳定组件。将不稳定组件放在该结构图的顶层是很有用的，因为这样我们就可以很容易地找出

箭头向上的依赖关系，而这些关系是违反 SDP（以及后面将会讨论的 ADP）的。

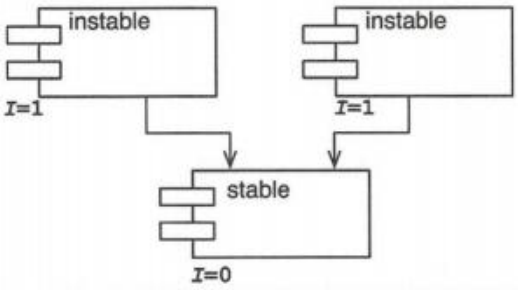


图 14.8：一个具有三个组件的系统的理想配置

下面再通过图 14.9 来看看违反 SDP 的情况：

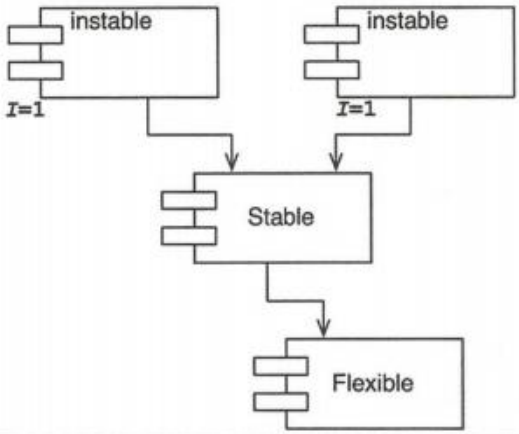


图 14.9：违反 SDP 的情况

在图 14.9 中，Flexible 是在设计中要确保其易于变更的组件，因此我们会希望 Flexible 是不稳定的。然而，Stable 组件的开发人员却引入了对 Flexible 组件的依赖。这种情况就违反了 SDP，因为 Stable 组件的 *I* 指标要远小于 Flexible 的 *I* 指标。这将导致 Flexible 组件的变更难度大大增加，因为对 Flexible 组件的任何修改都必须要考虑 Stable 组件及该组件自身存在的依赖关系。

如果想要修复这个问题，就必须要将 Stable 与 Flexible 这两个组件之间的依赖关系打破。为此，我们就需要了解这个依赖关系到底为什么会存在，这里假设是因为 Stable 组件中的某个类 U 需要使用 Flexible 组件中的一个类 C，如图

14.10 所示：

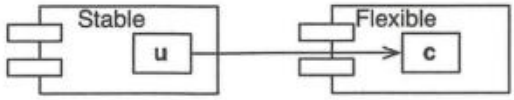


图 14.10：Stable 组件中的 U 使用了 Flexible 组件中的 C

我们可以利用 DIP 来修复这个问题。具体来说就是创造一个 UServer 组件，并在其中设置一个 US 接口类。然后，确保这个接口类中包含了所有 U 需要使用的函数，再让 C 实现这个接口，如图 14.11 所示。这样一来，我们就将从 Stable 到 Flexible 的这条依赖关系打破了，强迫这两个组件都依赖于 UServer。现在，UServer 组件会是非常稳定的（*I*=0），而 Flexibile 组件则会依然保持不稳定的状态（*I*=1），结构图中所有的依赖关系都流向 *I* 递减的方向了。

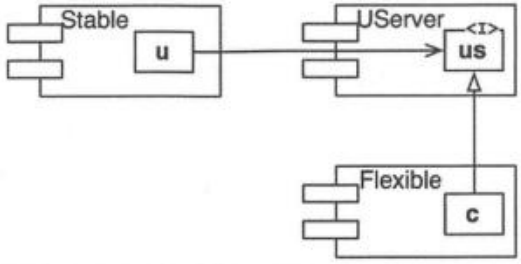


图 14.11：让 C 实现接口类 US

抽象组件

读者可能会觉得创造新组件（譬如上述例子中的 UService 组件，它其实只包含了一个接口类）这种做法挺奇怪的。因为这样的组件中几乎不包含任何可执行的代码！但事实上，这种做法在 C#或者 Java 这种静态类型语言中是非常普遍的，而且也必须这样做。因为这些抽象组件通常会非常稳定，可以被那些相对不稳定的组件依赖。

而当我们使用 Ruby 和 Python 这种动态类型语言时，这些抽象接口事实上并不存在，因此也就没有对它们的依赖。动态类型语言中的依赖关系是非常简单的，因为其依赖反转的过程并不需要声明和继承接口。

稳定抽象原则

一个组件的抽象化程度应该与其稳定性保持一致。

高阶策略应该放在哪里

在一个软件系统中，总有些部分是不应该经常发生变更的。这些部分通常用于表现该系统的高阶架构设计及一些策略相关的高阶决策。我们不想让这些业务决策和架构设计经常发生变更，因此这些代表了系统高阶策略的组件应该被放到稳定组件（I=0）中，而不稳定的组件（I=1）中应该只包含那些我们想要快速和方便修改的部分。

然而，如果我们将高阶策略放入稳定组件中，那么用于描述那些策略的源代码就很难被修改了。这可能会导致整个系统的架构设计难于被修改。如何才能让一个无限稳定的组件（I=0）接受变更呢？开闭原则（OCP）为我们提供了答案。这个原则告诉我们：创造一个足够灵活、能够被扩展，而且不需要修改的类是可能的，而这正是我们所需要的。哪一种类符合这个原则呢？答案是抽象类。

稳定抽象原则简介

稳定抽象原则（SAP）为组件的稳定性与它的抽象化程度建立了一种关联。一方面，该原则要求稳定的组件同时应该是抽象的，这样它的稳定性就不会影响到扩展性。另一方面，该原则也要求一个不稳定的组件应该包含具体的实现代码，这样它的不稳定性就可以通过具体的代码被轻易修改。

因此，如果一个组件想要成为稳定组件，那么它就应该由接口和抽象类组成，以便将来做扩展。如此，这些既稳定又便于扩展的组件可以被组合成既灵活又不会受到过度限制的架构。

将 SAP 与 SDP 这两个原则结合起来，就等于组件层次上的 DIP。因为 SDP 要求的是让依赖关系指向更稳定的方向，而 SAP 则告诉我们稳定性本身就隐含了对抽象化的要求，即依赖关系应该指向更抽象的方向。

然而，DIP 毕竟是与类这个层次有关的原则——对类来说，设计是没有灰色地带的。一个类要么是抽象类，要么就不是。SDP 与 SAP 这对原则是应用在组件层面上的，我们要允许一个组件部分抽象，部分稳定。

衡量抽象化程度

下面，假设 A 指标是对组件抽象化程度的一个衡量，它的值是组件中抽象类与接口所占的比例。那么：

- *Nc*：组件中类的数量。
- *Na*：组件中抽象类和接口的数量。
- *A*：抽象程度，*A*=*Na*÷*Nc*。

A 指标的取值范围是从 0 到 1，值为 0 代表组件中没有任何抽象类，值为 1 就意味着组件中只有抽象类。

主序列

现在，我们可以来定义组件的稳定性 *I* 与其抽象化程度 *A* 之间的关系了，具体如图 14.12 所示。在该图中，纵轴为 *A* 值，横轴为 *I* 值。如果我们将两个“设计良好”的组件绘制在该图上，那么最稳定的、包含了无限抽象类的组件应该位于左上角(0,1)，最不稳定的、最具体的组件应该位于右下角(1,0)。

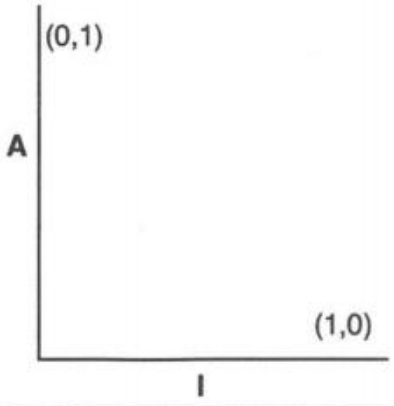


图 14.12：I/A 图

当然，不可能所有的组件都能处于这两个位置上，因为组件通常都有各自的稳定程度和抽象化程度。例如一个抽象类有时会衍生于另一个抽象类，这种情况是很常见的，而这个衍生过程就意味着某种依赖关系的产生。因此，虽然该组件是全抽象的，但它并不是完全稳定的，上述依赖关系降低了它的稳定程度。

既然不能强制要求所有的组件都处于(0,1)和(1,0)这两个位置上，那么就必须假设A/I图上存在着一个合理组件的区间。而这个区间应该可以通过排除法推导出来，也就是说，我们可以先找出那些组件不应该处于的位置（请参考图14.13）。

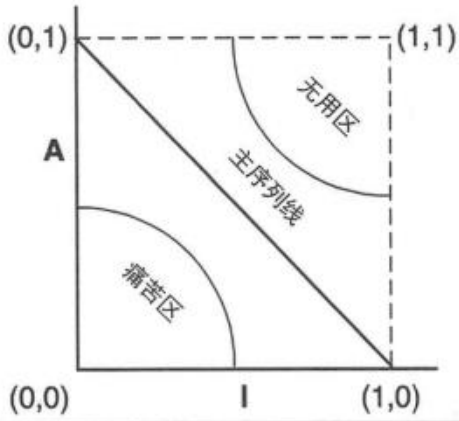


图14.13：排除区

痛苦区

在图14.13中，假设某个组件处于(0,0)位置，那么它应该是一个非常稳定但也非常具体的组件。这样的组件在设计上是不佳的，因为它很难被修改，这意味着该组件不能被扩展。这样一来，因为这个组件不是抽象的，而且它又由于稳定性的原因变得特别难以被修改，我们并不希望一个设计良好的组件贴近这个区域，因此(0,0)周围的这个区域被我们称为痛苦区（*zone of pain*）。

当然，有些软件组件确实会处于这个区域中，这方面的一个典型案例就是数据库的表结构（schema）。它在可变性上可谓臭名昭著，但是它同时又非常具体，并被非常多的组件依赖。这就是面向对象应用程序与数据库之间的接口这么难以管理，以及每次更新数据库的过程都那么痛苦的原因。

另一个会处于这个区域的典型软件组件是工具型类库。虽然这种类库的 *I* 指标为 1，但事实上通常是不可变的。例如 `String` 组件，虽然其中所有的类都是具体的，但由于它被使用得太过普遍，任何修改都会造成大范围的混乱，因此 `String` 组件只能是不可变的。

不可变组件落在(0,0)这一区域中是无害的，因为它们不太可能会发生变更。正因为如此，只有多变的软件组件落在痛苦区中才会造成麻烦，而且组件的多变性越强，造成的麻烦就会越大。其实，我们应该将多变性作为图 14.13 的第三个轴，这时图 14.13 所展示的便是多变性=1 时的情况，也就是最痛苦的切面。

无用区

现在我们来看看靠近(1,1)这一位置点的组件。该位置上的组件不会是我们想要的，因为这些组件通常是无限抽象的，但是没有被其他组件依赖，这样的组件往往无法使用。因此我们将这个区域称为*无用区*。

对于这个区域中的软件组件来说，其源码或者类中的设计问题通常是由于历史原因造成的。例如我们常常会在系统的某个角落里看到某个没有人实现的抽象类，它们一直静静地躺在那里，没有人使用。

同样的，落在无用区中的组件也一定会包含大量的无用代码。很明显，这类组件也不是我们想要的。

避开这两个区域

很明显，最多变的组件应该离上述两个区域越远越好。在图 14.13 中，我们可以将距离两个区域最远的点连成一条线，即从(1,0)连接到(0,1)。我将这条线称为*主序列线*（*main sequence*）。[1]

坐落于主序列线上的组件不会为了追求稳定性而被设计得“太过抽象”，也不会为了避免抽象化而被设计得“太过不稳定”。这样的组件既不会特别难以被修改，又可以实现足够的功能。对于这些组件来说，通常会有足够多的组件依赖于它们，

1　这里借用了天文学上的一个重要名词，请读者见谅。

这使得它们会具有一定程度的抽象，同时它们也依赖了足够多的其他组件，这又使得它一定会包含很多具体实现。

在整条主序列线上，组件所能处于最优的位置是线的两端。一个优秀的软件架构师应该争取将自己设计的大部分组件尽可能地推向这两个位置。然而，以我的个人经验来说，大型系统中的组件不可能做到完全抽象，也不可能做到完全稳定。所以我们只要追求让这些组件位于主序列线上，或者贴近这条线即可。

离主序列线的距离

接下来介绍最后一个指标：如果让组件位于或者靠近主序列是可取的目标，那么我们就可以创建一个指标来衡量一个组件距离最佳位置的距离。

- *D* 指标[1]：距离 $D=|A+I-1|$，该指标的取值范围是[0,1]。值为 0 意味着组件是直接位于主序列线上的，值为 1 则意味着组件在距离主序列最远的位置。

通过计算每个组件的 *D* 指标，就可以量化一个系统设计与主序列的契合程度了。另外，我们也可以用 *D* 指标大于 0 多少来指导组件的重构与重新设计。

除此之外，通过计算设计中所有组件的 *D* 指标的平均值和方差，我们还可以用统计学的方法来量化分析一个系统设计。对于一个良好的系统设计来说，*D* 指标的平均值和方差都应该接近于 0。其中，方差还可以被当作组件的“达标红线”来使用，我们可以通过它找出系统设计中那些不合常规的组件。

在图 14.14 中，我们可以看到大部分的组件都位于主序列附近，但是有些组件处于平均值的标准差（$Z=1$）以外。这些组件值得被重点分析，它们要么过于抽象但依赖不足，要么过于具体而被依赖得太多。

1 在我之前的文章中，曾经将这个指标叫作 D'，但是在这本书里显然没有理由这样做。

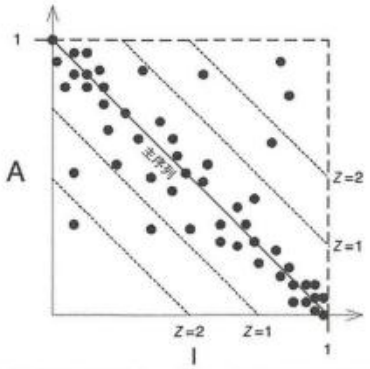


图 14.14：组件分散图

D 指标的另外一种用法是按时间来跟踪每个组件的值，下面用图 14.15 来做一个示范。在该图中可以看到，`Payroll` 组件在最近几次发布中累积了一些意外的对外依赖。图中的 *D*=0.1 是组件的达标红线，R2.1 这个值已经超出了红线范围，这就告诉我们现在值得花一些精力来找出这个组件偏离主序列线的原因了。

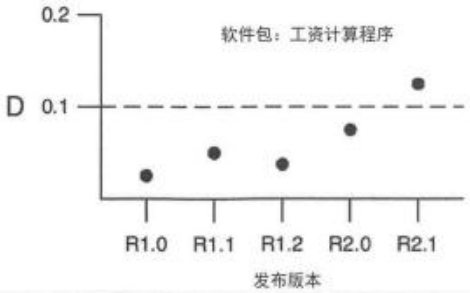


图 14.15：针对单组件的 *D* 值时间趋势图

本章小结

本章介绍了各种可用于依赖关系管理的指标，它们可以被用来量化分析某个系统设计与“优秀”设计模式之间的契合度。根据以往的经验，组件之间有些依赖关系是好的，有些依赖关系则是不好的，这些经验最后都会体现在这个设计模式中。当然，指标并不等同于真理，它只是对我们所定义标准的一个衡量。这些指标肯定是不完美的，但是我希望它们对读者有价值。

第 5 部分

软件架构

第 15 章

什么是软件架构

“架构”这个词给人的直观感受就充满了权力与神秘感，因此谈论架构总让人有一种正在进行责任重大的决策或者深度技术分析的感觉。毕竟，进阶到软件架构这一层次是我们走技术路线的人的终极目标。一个软件架构师总是给人一种权力非凡、广受尊敬的感觉，有哪个年轻的软件工程师没有梦想过成为一个软件架构师呢？

那么，究竟什么才是“软件架构”呢？软件架构师的工作内容究竟是什么？这项工作又是什么时候进行的呢？

首先，软件架构师自身需要是程序员，并且必须一直坚持做一线程序员，绝对不要听从那些说应该让软件架构师从代码中解放出来以专心解决高阶问题的伪建议。不是这样的！软件架构师其实应该是能力最强的一群程序员，他们通常会在自身承接编程任务的同时，逐渐引导整个团队向一个能够最大化生产力的系统设计方向前进。也许软件架构师生产的代码量不是最多的，但是他们必须不停地承接编程任务。如果不亲身承受因系统设计而带来的麻烦，就体会不到设计不佳所带来的痛苦，接着就会逐渐迷失正确的设计方向。

软件系统的架构质量是由它的构建者所决定的，软件架构这项工作的实质就是规划如何将系统切分成组件，并安排好组件之间的排列关系，以及组件之间互相通信的方式。

而设计软件架构的目的，就是为了在工作中更好地对这些组件进行研发、部署、运行以及维护。

> 如果想设计一个便于推进各项工作的系统，其策略就是要在设计中尽可能长时间地保留尽可能多的可选项。

上面这句话可能会让人很意外，也许你一直认为设计软件架构的目标应该是让一个系统能正确地工作。我们当然需要让系统正常工作，软件架构设计最高优先级的目标就是保持系统正常工作。

然而，一个软件系统的架构质量和该系统是否能正常工作的关系并不大，毕竟世界上有很多架构设计糟糕但是工作正常的软件系统。真正的麻烦往往并不会在我们运行软件的过程中出现，而是会出现在这个软件系统的开发、部署以及后续的补充开发中。

当然，这并不意味着好的软件架构对系统的行为就没有影响了，事实上架构在其中的角色还是很重要的。然而在这个方面，架构能起到的作用更多的时候是被动的，修饰性的，并不是主动的，更不是必不可少的。在系统的架构设计中，能影响系统行为的可选项少之又少。

软件架构设计的主要目标是支撑软件系统的全生命周期，设计良好的架构可以让系统便于理解、易于修改、方便维护，并且能轻松部署。软件架构的终极目标就是最大化程序员的生产力，同时最小化系统的总运营成本。

开发（Development）

一个开发起来很困难的软件系统一般不太可能会有一个长久、健康的生命周期，所以系统架构的作用就是要方便其开发团队对它的开发。

这意味着，不同的团队结构应该采用不同的架构设计。一方面，对于一个只有五个开发人员的小团队来说，他们完全可以非常高效地共同开发一个没有明确定义组件和接口的单体系统（monolithic system）。事实上，这样的团队可能会发现软件架构在早期开发中反而是一种障碍。这可能就是为什么许多系统都没有设计一个良好架构的原因，因为它们的开发团队起初都很小，不需要设计一些上层建筑来限制某些事情。

但另一方面，如果一个软件系统是由五个不同的团队合作开发的，而每个团队各自都有七个开发人员的话，不将系统划分成定义清晰的组件和可靠稳定的接口，开发工作就没法继续推进。通常，如果忽略其他因素，该系统的架构会逐渐演变成五个组件，一个组件对应一个团队。

当然，这种一个组件对应一个团队的架构不太可能是该系统在部署、运行以及维护方面的最优方案。但不管怎样，如果研发团队只受开发进度来驱动的话，他们的架构设计最终一定会倾向于这个方向。

部署（Deployment）

为了让开发成为有效的工作，软件系统就必须是可部署的。在通常情况下，一个系统的部署成本越高，可用性就越低。因此，实现一键式的轻松部署应该是我们设计软件架构的一个目标。

但很不幸，我们在系统的早期开发中很少会考虑部署策略方面的事情，这常常会导致一些易于开发、难于部署的系统架构。

例如，在系统的早期开发中，开发人员可能会决定采用某种“微服务架构”。这种架构的组件边界清晰，接口稳定，非常利于开发。但当我们实际部署这种系统时，就会发现其微服务的数量已经大到令人望而生畏，而配置这些微服务之间的连接以及启动时间都会成为系统出错的主要来源。

如果软件架构师早先就考虑到这些部署问题，可能就会有意地减少微服务的数量，采用进程内部组件与外部服务混合的架构，以及更加集成式的连接管理方式。

运行（Operation）

软件架构对系统运行的影响远不及它对开发、部署和维护的影响。几乎任何运行问题都可以通过增加硬件的方式来解决，这避免了软件架构的重新设计。

事实上，我们长期以来就一直目睹着这种情况一再发生。对于一个因架构设计糟糕而效率低下的系统，我们通常只需要增加更多的存储器与服务器，就能够让它圆满地完成任务。另外，硬件也远比人力要便宜，这也是软件架构对系统运行的影响远没有它对开发、部署、维护的影响那么深远的一个原因。

当然，这并不是说我们不应该为了让系统能更好地运转而优化软件的架构设计，这样做是应该的，只是基于投入/产出比的考虑，我们的优化重心应该更倾向于系统的开发、部署以及维护。

即使这样，软件架构在整个系统运行的过程中还发挥着另外一个重要作用，那

就是一个设计良好的软件架构应该能明确地反映该系统在运行时的需求。

也许我们可以换一个更好的说法，那就是设计良好的系统架构应该可以使开发人员对系统的运行过程一目了然。架构应该起到揭示系统运行过程的作用。具体来说，就是该架构应该将系统中的用例、功能以及该系统的必备行为设置为对开发者可见的一级实体，简化它们对于系统的理解，这将为整个系统的开发与维护提供很大的帮助。

维护（Maintenance）

在软件系统的所有方面中，维护所需的成本是最高的。满足永不停歇的新功能需求，以及修改层出不穷的系统缺陷这些工作将会占去绝大部分的人力资源。

系统维护的主要成本集中在“探秘”和“风险”这两件事上。其中，“探秘（spelunking）”的成本主要来自我们对于现有软件系统的挖掘，目的是确定新增功能或被修复问题的最佳位置和最佳方式。而“风险（risk）”，则是指当我们进行上述修改时，总是有可能衍生出新的问题，这种可能性就是风险成本。

我们可以通过精雕细琢的架构设计极大地降低这两项成本。通过将系统切分为组件，并使用稳定的接口将组件隔离，我们可以将未来新功能的添加方式明确出来，并大幅度地降低在修改过程中对系统其他部分造成伤害的可能性。

保持可选项

正如我们在之前章节中所说的，软件有行为价值与架构价值两种价值。这其中的第二种价值又比第一种更重要，因为它正是软件之所以“软”的原因。

软件被发明出来就是因为我们需要一种灵活和便捷的方式来改变机器的行为。而软件的灵活性则取决于系统的整体状况、组件的布置以及组件之间的连接方式。

我们让软件维持“软”性的方法就是尽可能长时间地保留尽可能多的可选项。

那么到底哪些选项是我们应该保留的？它们就是那些无关紧要的细节设计。

基本上，所有的软件系统都可以降解为策略与细节这两种主要元素。策略体现的是软件中所有的业务规则与操作过程，因此它是系统真正的价值所在。

而细节则是指那些让操作该系统的人、其他系统以及程序员们与策略进行交互，但是又不会影响到策略本身的行为。它们包括 I/O 设备、数据库、Web 系统、服务器、框架、交互协议等。

软件架构师的目标是创建一种系统形态，该形态会以策略为最基本的元素，并让细节与策略脱离关系，以允许在具体决策过程中推迟或延迟与细节相关的内容。

例如，

- 在开发的早期阶段应该无须选择数据库系统，因为软件的高层策略不应该关心其底层到底使用哪一种数据库。事实上，如果软件架构师足够小心，软件的高层策略甚至可以不用关心该数据库是关系型数据库，还是分布式数据库，是多级数据库，还只是一些文本文件而已。
- 在开发的早期阶段也不应该选定使用的 Web 服务，因为高层策略并不应该知道自己未来要以网页形式发布。如果高层策略能够与 HTML、AJAX、JSP、JSF 或任何 Web 开发技术脱钩，那么我们就可以将对 Web 系统的选择推迟到项目的最后阶段。事实上，很有可能我们压根不需要考虑这个系统到底是不是以网页形式发布的。
- 在开发的早期阶段不应该过早地采用 REST 模式，因为软件的高层策略应该与外部接口无关。同样的，我们也不应该过早地考虑采用微服务框架、SOA 框架等。再说一遍，软件的高层策略压根不应该跟这些有关。
- 在开发的早期阶段不应过早地采用依赖注入框架（dependency injection framework），因为高层策略不应该操心如何解析系统的依赖关系。

说到这里，我想读者应该明白我的意思了。如果在开发高层策略时有意地让自己摆脱具体细节的纠缠，我们就可以将与具体实现相关的细节决策推迟或延后，因为越到项目的后期，我们就拥有越多的信息来做出合理的决策。

同时，这样做还可以让我们有机会做不同的尝试。例如。如果我们现在手里有

一部分与数据库无关的高层策略，那么我们就可以用不同的数据库来做实验，以检验该系统与不同数据库之间的适应性和性能。类似的情况也适用于各种 Web 框架，甚至 Web 这种发布形式本身。

另外，我们保留这些可选项的时间越长，实验的机会也就越多。而实验做得越多，我们做决策的时候就能拥有越充足的信息。

那么如果其他人已经替我们做出了决策呢？譬如说，我们的公司已经指定了某个数据库，或某种 Web 服务，或某个框架，这时应该怎么办？通常一个优秀的软件架构师会假装这些决策还没有确定，并尽可能长时间地让系统有推迟或修改这些决策的能力。

> 一个优秀的软件架构师应该致力于最大化可选项数量。

设备无关性

如果想要找反映这方面思想的例子，我们还得先回到 20 世纪 60 年代。由于当时的计算机行业还处于萌芽阶段，大部分程序员都来自数学专业，或者是其他工程类专业（当时超过三分之一的程序员是女性）。

当时，我们曾经犯过很多错误，而且还没有人知道那些是错误。当然了，那时候我们怎么可能知道？

其中一个错误就是将代码与 I/O 设备直接紧密地绑定在一起。当时，如果我们需要用打印机打印东西，就得专门写一段 I/O 指令来操作打印机，因此我们的代码是依赖于设备的。

例如，当我们要写一段要在电传打印机上输出的 PDP-8 程序时，需要用到像下面这样一组机器指令：

```
PRTCHR, 0
        TSF
        JMP .-1
        TLS
```

```
JMP I PRTCHR
```

这里的 PRTCHR 是电传打印机上一段用来打印字符的子程序。首语句中的 0 是存储其返回地址用的（这里就不要细究这些了）。下来是 TSF 指令，它的作用是告诉电传打印机如果准备就绪，就跳过下一指令。如果电传打印机处于繁忙状态，就继续执行 JMP.-1 指令，也就是再跳转回 TSF 指令。一旦电传打印机处于就绪状态，TSF 就会跳转到 TLS 指令，该指令会将 A 寄存器中保存的要打印的字符发送给电传打印机。随后，JMP I PRTCHR 指令会将程序返回给调用方。

一开始，这一策略工作起来完全没有问题。如果我们需要从读卡器中读取卡片，我们就直接用代码与读卡器进行交互。如果我们需要在卡上打孔，就写一段代码直接控制打卡的过程。整套程序运行得非常完美。我们当时怎么会知道这是一个错误呢？

然而，管理大量的卡片是一件很麻烦的事。这些卡片可能会出现丢失、损坏、旋转、排序错误等问题。各部分的卡片都有可能丢失或混入多余的卡片，保持数据的一致性是当时的一大难题。

后来就出现了磁带这种解决方案。它允许我们将原本打在卡片上的图像存储在磁带上。如果磁带不小心掉在地上，不会出现顺序被打乱的问题，我们也不会因此意外丢失记录，或者处理磁带时意外插入空白记录。显然，磁带是更安全的选择，而且它的读取和写入也更快，同时也很容易进行备份。

但不幸的是，我们当时所有的软件都是用于直接操作读卡器和打卡器的。为了让这些软件改用磁带，我们不得不花很大的力气重新修改代码。

到了 20 世纪 60 年代末期，我们已经吸取了这个教训，并为此提出了设备无关性这个概念。当时的操作系统会将 I/O 设备抽象成打孔卡那样的，处理一条条记录的标准软件函数。我们写的程序会通过调用操作系统提供的服务来与抽象的记录处理函数进行交互。而系统运行人员可以将操作系统的抽象设备与具体的读卡器、磁带读取器以及其他类似的设备进行对接。

这样一来，同一段程序不经任何修改就既可以读/写卡片，也可以读/写磁带。开闭原则（OCP）此时就诞生了（当然，那时候还不叫这个名字）。

垃圾邮件

20 世纪 60 年代末期，我曾经在一家为客户打印群发垃圾邮件的公司工作。当时，客户会将一条条与消费者名字和地址相关的记录存储在磁带中并寄给我们，我们则负责编写程序为他们打印个人化的广告。

相信下面这些邮件读者一定不陌生。

Hello，马丁先生：

恭喜！

您是 Witchwood Lane 上唯一被选中参加我们仅有一次的特惠活动……

客户会给我们寄来一大卷信纸，其中的姓名和地址留空，其他文字都已经填好。我们的程序需要从磁带上读取姓名、地址等信息，然后将这些信息精确地打印在信纸上的对应位置。

这样的每一卷信纸里面有几千封信，重量近 500 磅，而且通常有数百卷之多，我们必须一封一封地打印。

起初，我们使用的是 IBM 360 自带的单行打印机，它每个工作日可以打印几千张。但是，当时 IBM 360 每个月的租金要几万美金，成本太高了。

这时候，我们只需要让操作系统放弃单行打印机，改用磁带即可，我们的程序不需要做任何的改动，因为它们使用的是操作系统提供的抽象 I/O 设备接口。

而且 IBM 360 机器每 10 分钟就可以写满一卷磁带——这一时间足够单行打印机打印几卷信纸了。然后这些磁带可以从计算机上取下，装载到离线打印机上进行离线打印。当时我们有五台这样的打印机，它们可以 7×24 小时不停地工作，每周可以打印几十万封信。

设备无关性的价值真是太巨大了！它使我们的程序不再需要关心具体使用的 I/O 设备。这样一来，我们可以用本地连接的打印机来调试程序，随后将它“打印”到

磁带卷上，并放到离线打印机上进行批量打印。

这段程序是有架构设计的，并且在设计中实现了高层策略与底层实现细节的分离。其策略部分负责格式化姓名和地址，细节部分负责操作具体的 I/O 设备。而我们具体采用哪个设备的决策是最后才做出的。

物理地址寻址

20 世纪 70 年代早期，我曾为本地卡车工会编写过一套大型的账务系统。当时，Agent、Employer、Member 这些记录都被存储在一块 25MB 大小的磁盘上。由于不同的记录尺寸不同，所以我们将磁盘的前几个柱面（cylinder）按 Agent 记录的大小格式化每个扇区，中间的按 Employer 记录的大小格式化，最后几个柱面按照 Member 记录的大小格式化。

当时我们编写的软件需要知道硬盘的具体结构。它知道每个硬盘包含 200 个柱面，10 个磁头，每个柱面每个磁头有几十个扇区。它也知道哪些柱面上包含的是 Agent 记录，哪些柱面上包含的是 Employer 和 Member 记录，我们对所有的这些都进行了硬编码。

另外，我们还在磁盘上保留了一个索引，以方便后续的记录查询。该索引也是通过一个特别的格式被存储到磁盘上的。譬如说，Agent 记录的索引中每条记录包括 Agent 的 ID，以及对应的柱面号码、磁头号码、扇区号码。Employer 和 Member 的索引也有类似的结构。其中，Member 记录用一种双向链表结构存储在磁盘上。每条 Member 记录都会包含前一个和后一个 Member 记录所在的柱面号码、磁头号码、扇区号码。

在这种情况下，如果我们升级新硬盘会发生什么呢？新硬盘可能会有更多的磁头，更多的柱面，或是每个柱面有更多的扇区。这时候，我们就必须编写一个特殊的程序从旧磁盘读取数据，并将其写入新磁盘，同时换掉柱面、磁头、扇区的值。另外，我们还要修改代码中所有硬编码的部分——这样的代码到处都是！毕竟我们所有的业务逻辑都和柱面、磁头、扇区的分配方案紧密地耦合在了一起。

直到有一天，一位更有经验的程序员加入了我们团队。当他看到我们的程序实现逻辑时差点吐血，就像见到外星人一样盯着我们看了半天。随后，他温柔地建议我们改用相对地址方式来寻址。

这位聪明的同事建议我们将磁盘当成一个扇区的线性队列来处理，用一串连续的整数来对每个扇区进行寻址。然后，我们可以编写一个针对磁盘物理结构的转换程序，以便将这些相对地址在线转换为柱面、磁头、扇区的号码。

幸运的是，我们采纳了他的建议。我们修改了系统的高层策略，使其与磁盘的物理结构脱钩。这样一来，我们就可以将具体选择哪种磁盘的决策从该应用程序中分离出来。

本章小结

在本章中，我们用两个小故事示范了一些架构师们普遍会采用的设计原则。优秀的架构师会小心地将软件的高层策略与其底层实现隔离开，让高层策略与实现细节脱钩，使其策略部分完全不需要关心底层细节，当然也不会对这些细节有任何形式的依赖。另外，优秀的架构师所设计的策略应该允许系统尽可能地推迟与实现细节相关的决策，越晚做决策越好。

第16章

独立性

正如我们之前所述，一个设计良好的软件架构必须支持以下几点。

- 系统的用例与正常运行。
- 系统的维护。
- 系统的开发。
- 系统的部署。

用例

我们先来看第一个支持目标：用例[1]。我们认为一个系统的架构必须能支持其自身的设计意图。也就是说，如果某系统是一个购物车应用，那么该系统的架构就必须非常直观地支持这类应用可能会涉及的所有用例。事实上，这本来就是架构师们首先要关注的问题，也是架构设计过程中的首要工作。软件的架构必须为其用例提供支持。

然而，正如我们前面所讨论的，一个系统的架构对其行为并没有太大的影响。虽然架构也可以限制一些行为选项，但这种影响所涉及的范围并不大。一个设计良好的架构在行为上对系统最重要的作用就是明确和显式地反映系统设计意图的行为，使其在架构层面上可见。

譬如说，一个架构优良的购物车应用看起来就该像是一个购物车应用。该系统的主要用例会在其系统结构上明确可见。开发人员将不需要在系统中查找系统所应有的行为，因为这些行为在系统顶层作为主要元素已经是明确可见的了，这些元素会以类、函数或模块的形式在架构中占据明显位置，它们的名字也能够清晰地描述对应的功能。

在第 21 章“尖叫的软件架构”中，我们还会更详细地解释这部分内容。

1 用例（use case），或译作使用案例。它是软件工程或系统工程中对系统如何响应外界请求的描述，是一种通过用户的使用场景来获取需求的技术。每个用例会提供一个或多个场景，这些场景说明了系统是如何和最终用户或其他系统进行交互的，也就是谁可以用系统做什么，从而获得一个明确的业务目标。编写用例时要避免使用技术术语，而应该用最终用户或者领域专家的语言。总而言之，用例一般是由软件开发者和最终用户共同创作的。——编者注

运行

架构在支持系统运行方面扮演着更实际的角色。如果某个系统每秒要处理100 000个用户，该系统的架构就必须能支持这种级别的吞吐量和响应时间。同样的，如果某个系统要在毫秒级的时间内完成对大数据仓库的查询，那么该系统的架构也必须能支持这类操作。

对一些系统来说，这意味着它的架构应该支持将其计算部分拆分成一系列小型服务，然后让它们并行运行在不同的服务器上。而在另一些系统中，采用一堆轻量级线程，然后让这些线程共享一个运行在单处理器上的进程的地址空间。还有一些系统，它们可能只是一组运行在独立地址空间内的进程。甚至有些系统设计为一个单进程的单体程序就够了。

虽然看起来有点奇怪，但上述问题的决策的确也应该属于一个优秀的架构师为我们保留的可选项之一。毕竟一个按照单体模式编写的系统，它依赖的必然是单体结构，之后再想把它改造成多进程、多线程或微服务模式可就没有那么容易了。相比之下，如果该系统的架构能够在其组件之间做一些适当的隔离，同时不强制规定组件之间的交互方式，该系统就可以随时根据不断变化的运行需求来转换成各种运行时的线程、进程或服务模型。

开发

系统的架构在支持开发环境方面当然扮演着重要的角色，我们在这里可以引述一下康威定律：

> 任何一个组织在设计系统时，往往都会复制出一个与该组织内沟通结构相同的系统。

一个由多个不同目标的团队协作开发的系统必须具有相应的软件架构。这样，这些团队才可以各自独立地完成工作，不会彼此干扰。这就需要恰当地将系统切分

为一系列隔离良好、可独立开发的组件。然后才能将这些组件分配给不同的团队，各自独立开发。

部署

一个系统的架构在其部署的便捷性方面起到的作用也是非常大的。设计目标一定是实现“立刻部署”。一个设计良好的架构通常不会依赖于成堆的脚本与配置文件，也不需要用户手动创建一堆“有严格要求”的目录与文件。总而言之，一个设计良好的软件架构可以让系统在构建完成之后立刻就能部署。

同样的，这些也需要通过正确地划分、隔离系统组件来实现，这其中包括开发一些主组件，让它们将整个系统黏合在一起，正确地启动、连接并监控每个组件。

保留可选项

一个设计良好的架构应该充分地权衡以上所述的所有关注点，然后尽可能地形成一个可以同时满足所有需求的组件结构。这说起来还挺容易的，不是吗？

事实上，要实现这种平衡是很困难的。主要问题是，我们在大部分时间里是无法预知系统的所有用例的，而且我们也无法提前预知系统的运行条件、开发团队的结构，或者系统的部署需求。更糟糕的是，就算我们能提前了解这些需求，随着系统生命周期的演进，这些需求也会不可避免地发生变化。总而言之，事实上我们想要达到的目标本身就是模糊多变的。真实的世界就这样。

然而，我们还是可以通过采用一些实现成本较低的架构原则来做一些事情的。虽然我们没有清晰的目标，但采用一些原则总是有助于提前解决一些平衡问题。通过遵守这些原则可以帮助我们正确地将系统划分为一些隔离良好的组件，以便尽可能长时间地为我们的未来保留尽可能多的可选项。

一个设计良好的架构应该通过保留可选项的方式，让系统在任何情况下都能方便地做出必要的变更。

按层解耦

从用例的角度来看，架构师的目标是让系统结构支持其所需要的所有用例。但是问题恰恰是我们无法预知全部的用例。好在架构师应该还是知道整个系统的基本设计意图的。也就是说，架构师应该知道自己要设计的是一个购物车系统，或是运输清单系统，还是订单处理系统。所以架构师可以通过采用单一职责原则（SRP）和共同闭包原则（CCP），以及既定的系统设计意图来隔离那些变更原因不同的部分，集成变更原因相同的部分。

哪些部分的变更原因是不同的呢？这在有些情况下是很显而易见的。譬如，用户界面的变更原因肯定和业务逻辑是不相关的，而业务用例则通常在两边都存在着相关的元素。所以很显然，优秀的架构师应该会将用例的 UI 部分与其业务逻辑部分隔离，这样这两部分就既可以各自进行变更，也能保证用例的完整清晰。

而业务逻辑则既可以是与应用程序紧密相关的，也可以是更具有普适性的。例如，对输入字段的校验是一个与应用程序本身紧密相关的业务逻辑。相反，计算账户利息以及清点库存则是一个与具体领域更为相关的业务逻辑。这两种不同的业务逻辑通常有着不同的变更速率和变更原因——它们应该被相互隔离，以方便各自的变更。

至于数据库，以及其所采用的查询语言，甚至表结构，这些都是系统的技术细节信息，它们与业务规则或 UI 毫无关系。这就意味着它们的变更原因、变更速率必然与系统的其他方面各不相同。因此，架构师也应该将它们与系统其他部分隔离，以方便各自的变更。

这样一来，我们就发现了一个系统可以被解耦成若干个水平分层——UI 界面、应用独有的业务逻辑、领域普适的业务逻辑、数据库等。

用例的解耦

接下来，还有什么不同原因的变更呢？答案正是这些用例本身！譬如说，添加新订单的用例与删除订单的用例在发生变更的原因上几乎肯定是不同的，而且发生变更的速率也不同。因此，我们按照用例来切分系统是非常自然的选择。

与此同时，这些用例也是上述系统水平分层的一个个垂直切片。每个用例都会用到一些 UI、特定应用的业务逻辑、应用无关的业务逻辑以及数据库功能。因此，我们在将系统水平切分成多个分层的同时，也在按用例将其切分成多个垂直切片。

为了实现这样的解耦，我们应该将增加订单这个用例的 UI 与删除订单用例的 UI 分开。而且，对业务逻辑的部分、数据库的部分，也要做同样的事情，将其按照用例进行垂直切分。

由此，我们可以总结出一个模式：如果我们按照变更原因的不同对系统进行解耦，就可以持续地向系统内添加新的用例，而不会影响旧有的用例。如果我们同时对支持这些用例的 UI 和数据库也进行了分组，那么每个用例使用的就是不同面向的 UI 与数据库，因此增加新用例就更不太可能会影响旧有的用例了。

解耦的模式

现在我们来想想所有的这些解耦动作对架构设计的第二个目标——系统运行——究竟有什么意义。如果不同面向之间的用例得到了良好的隔离，那么需要高吞吐量的用例就和需要低吞吐量的用例互相自然分开了。如果 UI 和数据库的部分能从业务逻辑分离出来，那么它们就可以运行在不同的服务器上。而且需要较大带宽的应用也可以在多个服务器上运行多个实例。

总而言之，这种按用例解耦的动作是有利于系统运行的。然而出于系统运行效率的考虑，我们的解耦动作还应该注意选择恰当的模式。譬如，为了在不同的服务器上运行，被隔离的组件不能依赖于某个处理器上的同一个地址空间，它们必须是

独立的服务，然后通过某种网络来进行通信。

许多架构师将上面这种组件称为“服务”或“微服务”，至于是前者还是后者，往往取决于某些非常模糊的代码行数阈值。对于这种基于服务来构建的架构，架构师们通常称之为面向服务的架构（service-oriented architecture）。

如果因为这里提到了 SOA 这个概念而引起了某些读者的警觉，请不用担心，我在这里并没有鼓吹 SOA 是一种最佳的软件架构，或者微服务就是未来的潮流。我只是认为有时候我们必须把组件切割到服务这个应用层次。

请记住，一个设计良好的架构总是要为将来多留一些可选项，这里所讨论的解耦模式也是这样的可选项之一。

接下来，在我们继续深入探讨这个话题之前，先回过头来看看其他两个设计目标。

开发的独立性

我们进行架构设计的第三个目标是支持系统的开发。很显然，当系统组件之间被高度解耦之后，开发团队之间的干扰就大大减少了。譬如说，如果系统的业务逻辑与其 UI 无关，那么专注于 UI 开发的团队就不会对专注于业务逻辑开发的团队造成多大的影响。同样的，如果系统的各个用例之间相互隔离，那么专注于 `addOrder` 用例的团队就不太可能干扰到负责 `deleteOrder` 用例的团队。

只要系统按照其水平分层和用例进行了恰当的解耦，整个系统的架构就可以支持多团队开发，不管团队组织形式是分功能开发、分组件开发、分层开发，还是按照别的什么变量分工都可以。

部署的独立性

这种按用例和水平分层的解耦也会给系统的部署带来极大的灵活性。实际上，

如果解耦工作做得好，我们甚至可以在系统运行过程中热切换（hot-swap）其各个分层实现和具体用例。在这种情况下，我们增加新用例就只需要在系统中添加一些新的 jar 文件，或启动一些服务即可，其他部分将完全不受影响。

重复

架构师们经常会钻进一个牛角尖——害怕重复。

当然，重复在软件行业里一般来说都是坏事。我们不喜欢重复的代码，当代码真的出现重复时，我们经常会感到作为一个专业人士，自己是有责任减少或消除这种重复的。

但是重复也存在着很多种情况。其中有些是真正的重复，在这种情况下，每个实例上发生的每项变更都必须同时应用到其所有的副本上。重复的情况中也有一些是假的，或者说这种重复只是表面性的。如果有两段看起来重复的代码，它们走的是不同的演进路径，也就是说它们有着不同的变更速率和变更缘由，*那么这两段代码就不是真正的重复*。等我们几年后再回过头来看，可能就会发现这两段代码是非常不一样的了。

现在，我们假设某系统中有两个用例在屏幕展现形式上非常类似。每当这种时候，架构师们就很可能非常想复用同一段代码来处理它们的屏幕展示。那么，我们到底是否应该这样做呢？这里是真正的重复，还只是一种表面性的重复？

恐怕这里很可能只是表面性的重复。随着时间推移，这两个用例的屏幕展示功能可能会各自演变，最终很可能完全不同。正是由于这样的原因，我们必须加倍小心地避免让这两个用例复用同一段代码，否则，未来再想将它们分开会面临很大的挑战。

当我们按用例垂直切分系统时，这样的问题会经常出现。我们经常遇到一些不同的用例为了上述原因被耦合在了一起。不管是因为它们展现形式类似，还是使用了相似的语法、相似的数据库查询/表结构等，总之，我们一定要小心避免陷入对任何重复都要立即消除的应激反应模式中。一定要确保这些消除动作只针对那些真正

意义上的重复。

同样的道理，当我们对系统进行水平分层时，也可能会发现某个数据库记录的结构和某个屏幕展示的数据接口非常相似。我们可能也会为了避免再创建一个看起来相同的视图模型并在两者之间复制元素，而选择直接将数据库记录传递给 UI 层。我们也一定要小心，这里几乎肯定只是一种表面性的重复。而且，另外创建一个视图模型并不会花费太多力气，这可以帮助我们保持系统水平分层之间的隔离。

再谈解耦模式

让我们再回到解耦模式的问题上来。按水平分层和用例解耦一个系统有很多种方式。例如，我们可以在源码层次上解耦、二进制层次上解耦（部署），也可以在执行单元层次上解耦（服务）。

- **源码层次**：我们可以控制源代码模块之间的依赖关系，以此来实现一个模块的变更不会导致其他模块也需要变更或重新编译（例如 Ruby Gem）。
  在这种解耦模式下，系统所有的组件都会在同一个地址空间内执行，它们会通过简单的函数调用来进行彼此的交互。这类系统在运行时是作为一个执行文件被统一加载到计算机内存中的。人们经常把这种模式叫作单体结构。
- **部署层次**：我们可以控制部署单元（譬如 jar 文件、DLL、共享库等）之间的依赖关系，以此来实现一个模块的变更不会导致其他模块的重新构建和部署。
  在这种模式下，大部分组件可能还是依然运行在同一个地址空间内，通过彼此的函数调用通信。但有一些别的组件可能会运行在同一个处理器下的其他进程内，使用跨进程通信，或者通过 socket 或共享内存进行通信。这里最重要的是，这些组件的解耦产生出许多可独立部署的单元，例如 jar 文件、Gem 文件和 DLL 等。
- **服务层次**：我们可以将组件间的依赖关系降低到数据结构级别，然后仅通过网络数据包来进行通信。这样系统的每个执行单元在源码层和二进制层都会是一个独立的个体，它们的变更不会影响其他地方（例如，常见的服

务或微服务就都是如此的）。

现在，我们要问的是究竟哪个模式是最好的呢？

答案是，在项目早期很难知道哪种模式是最好的。事实上，随着项目的逐渐成熟，最好的模式可能会发生变化。

例如，我们不难想象，一个在某台服务器上运行良好的程序发展到一定程度，可能就会需要将其某些组件迁移到其他服务器上才能满足运行要求。当该系统只运行在一台服务器上时，我们进行源码层次的解耦就已经足够了。但在这之后，我们可能需要进行部署单元层次的解耦，甚至服务层次的解耦。

另一个解决方案（似乎也是目前最流行的方案）是，默认就采用服务层次的解耦。这种做法的问题主要在于它的成本很高，并且是在鼓励粗粒度的解耦。毕竟，无论微服务有多么“微”，其解耦的精细度都可能是不够的。

服务层次解耦的另一个问题是不仅系统资源成本高昂，而且研发成本更高。处理服务边界不仅非常耗费内存、处理器资源，而且更耗费人力。虽然内存和处理器越来越便宜，但是人力成本可一直都很高。

通常，我会倾向于将系统的解耦推行到某种一旦有需要就可以随时转变为服务的程度即可，让整个程序尽量长时间地保持单体结构，以便给未来留下可选项。

在这种方式下，系统最初的组件隔离措施都是做在源码层次上的，这样的解耦可能在整个项目的生命周期里已经足够了。然而，如果部署和开发方面有更高的需求出现，那么将某些组件解耦到部署单元层次就可能够了，起码能撑上一阵。

当然，随着系统在开发、部署、运行各方面所面临的问题持续增加，我们应该挑选一下可以将哪些可部署单元转化为服务，并且逐渐将系统向这个方向转变。

而随着时间的流逝，系统的运维需求可能又会降低。之前需要进行服务层次解耦的系统可能现在只需要进行部署层次或源码层次的解耦就够了。

一个设计良好的架构应该能允许一个系统从单体结构开始，以单一文件的形式部署，然后逐渐成长为一组相互独立的可部署单元，甚至是独立的服务或者微服务。

最后还能随着情况的变化，允许系统逐渐回退到单体结构。

并且，一个设计良好的架构在上述过程中还应该能保护系统的大部分源码不受变更影响。对整个系统来说，解耦模式也应该是一个可选项。我们在进行大型部署时可以采用一种模式，而在进行小型部署时则可以采用另一种模式。

本章小结

是的，要达到上述要求难度不小。我并没有说系统的部署模式就一定要是某种简单的配置项（虽然在某些情况下的确应该这样做）。这里的主要观点认为，一个系统所适用的解耦模式可能会随着时间而变化，优秀的架构师应该能预见这一点，并且做出相应的对策。

第 17 章

划分边界

软件架构设计本身就是一门划分边界的艺术。边界的作用是将软件分割成各种元素，以便约束边界两侧之间的依赖关系。其中有一些边界是在项目初期——甚至在编写代码之前——就已经划分好，而其他的边界则是后来才划分的。在项目初期划分这些边界的目的是方便我们尽量将一些决策延后进行，并且确保未来这些决策不会对系统的核心业务逻辑产生干扰。

正如我们之前所说，架构师们所追求的目标是最大限度地降低构建和维护一个系统所需的人力资源。那么我们就需要了解一个系统最消耗人力资源的是什么？答案是系统中存在的耦合——尤其是那些过早做出的、不成熟的决策所导致的耦合。

那么，怎样的决策会被认为是过早且不成熟的呢？答案是那些决策与系统的业务需求（也就是用例）无关。这部分决策包括我们要采用的框架、数据库、Web 服务器、工具库、依赖注入等。在一个设计良好的系统架构中，这些细节性的决策都应该是辅助性的，可以被推迟的。一个设计良好的系统架构不应该依赖于这些细节，而应该尽可能地推迟这些细节性的决策，并致力于将这种推迟所产生的影响降到最低。

几个悲伤的故事

下面要讲的是一个来自 P 公司的悲伤的故事，我们在这里是将它作为一个草率决策的反面案例给大家展示的。在 20 世纪 80 年代，P 公司的创始团队开发了一个单体结构的简单桌面应用。然后，产品获得了极大的成功，并在 20 世纪 90 年代成功地成长为一个广为人知的桌面 GUI 应用。

然而到了 20 世纪 90 年代末期，Web 卷起了一股浪潮，突然间每个公司都在推出自己的 Web 解决方案，P 公司自然也不能置身事外。P 公司的客户吵闹着要求它提供一个 Web 版的产品。为了应对这种需求，该公司雇用了一群二十几岁的 Java 程序员，开始着手将他们的产品 Web 化。

结果，这群 Java 小子满脑子朝思暮想的就是如何将大规模服务器集群应用起来，

所以他们采用了一个三层的富“架构”[1]，将系统的各层应用分布到一个大型服务集群中，这样一来，GUI、中间件和数据库自然就都要运行在不同的服务器上。

这帮程序员在开发初期就决定系统中所有领域对象都要有三份实例：GUI 层一份、中间件层一份、数据库层一份。而由于这些实例要运行在不同的机器上，于是一套完整的跨处理器、跨层通信的富系统被设计了出来。该系统各层之间的函数调用都会被转变为对象，这些对象在经过序列化和编码处理之后进行网络传输。

现在假设我们想要增加一个特别简单的功能，为现有的记录添加一个字段。在上述情况下，这个字段必须被同步添加到系统所有三个分层的类，以及几个用于跨层通信的消息结构中。由于数据是双向流动的，这意味着我们要为此设计四个传输协议。而又由于每个协议都有各自的发送方和接收方，所以我们总共需要实现八个传输协议处理函数。总结一下，我们需要做的是同时构建三个可执行文件，每个文件都要包含三个变更过的业务对象、四个新的消息结构以及八个新的处理函数。

这就是我们添加一个最简单的功能所需要做的事情。读者可以想想所有这些新增对象要进行的初始化、序列化、编码和反编码、消息构建和解析、socket 通信、超时管理、重试情况处理等过程，我们做所有的这些事情就只是为了完成一点小小的功能吗？这代价未免太大了点。

当然，这帮程序员在开发过程中实际上是没有大型服务器集群可以用的。他们其实仍然是在一台机器上运行那三个可执行文件，而且就这样开发了几年时间。然而，即使这个系统只在一台机器上执行，也还是要经历所有这些对象的初始化、序列化、编码与反编码、消息的构建和解析、socket 通信等过程的，但他们直到最后还是坚信自己的架构是正确的。

这里最讽刺的是，P 公司从来就没有销售过一个需要服务器集群的系统，他们所有曾经部署过的系统都是在一台机器上完成的。然后在这台机器上，系统中所有的三个可执行文件仍然继续着这些对象初始化、序列化、编码与反编码、消息的构建与解析、socket 通信等不必要的工作，就是为了适应一个并不存在且也永远不会

1 这里将“架构”用引号括起来是因为这三层结构并不是真正的架构，而是一种部署拓扑，而后者恰恰是一个好的架构所要延后的决策之一。

存在的集群环境。

这个故事的悲剧之处在于，软件架构师通过一个草率的决定无谓地将开发成本放大了数倍之多。

而且，P 公司的故事绝不是个案，我在很多地方都看到过这个故事的各种版本。

接下来，还有比 P 公司更糟糕的故事。

我们再来看一下 W 公司的故事。该公司有一项管理其租赁车队的本地业务，他们最近招聘了一位“架构师”来控制他们目前的软件开发成本。而且据说“控制”这个词就是这家伙的中间名[1]。总之这位“架构师”来了之后很快做了判断，认为运作这个小小的业务需要的是一个全套的、企业级的、**面向服务的“架构”**。于是，他就创建了一个巨大的域模型，其中包含了该业务所涉及的所有“对象”，并设计了一整套服务来管理这些对象，差点将所有开发人员逼疯。在他这套“架构”中，如果我们想在销售记录中添加一个联系人，提供名字、地址和电话号码，就必须先访问 `ServiceRegistry`，查询 `ContactService` 的 ID。然后需要发送一条 `CreateContact` 消息给 `ContactService`。当然，这个消息结构有几十个字段，每一个字段都需要有效的数据来填充——这些数据是普通程序员无法访问的，他们手里只有名字、地址和电话号码。只有在伪造数据之后，程序员才能将新建的 Contact 记录 ID 填入 `UpdateContact` 消息中，最后还要发送给 `SaleRecordService`。

当然，为了测试这一切，我们还必须将所有的服务全部运行起来，同时还要运行消息总线、BPel 服务器等。更糟糕的是，这些消息还会在每个服务之间传递时出现延迟，需要一个队列接一个队列地等待。

如果还需要添加新功能的话，读者可以想象一下所有这些服务之间的耦合关系、那些需要修改的大量 WSDL 文件以及需要进行的重新部署。

说真的，地狱看起来也不过如此吧！

1　所谓中间名，实际上是英语国家的一个习俗，他们在姓和名之间有时候会加一个词，通常这个词有一些特殊用意，比如彰显母系血统、纪念某个祖先，或者强调自己的某种荣耀，作者在这里引申的是第三种用意，做讽刺之用。——编者注

按照服务来组织一个软件系统的结构这件事本身并没有什么问题，W 公司的错误主要在于太过草率地采用和强制推广了一套号称是SOA体系的工具——也就是说，他们过于草率地采用了一整套域对象服务体系。这个错误让该公司大量的人力都耗费在了实现这个所谓的 SOA 架构上。

我们还可以一个接一个地描述很多这类架构设计失败的案例，但还是先来讲一些成功的案例吧。

FitNesse

我和我儿子 Micah 在 2001 年创立了一家叫 `FitNesse` 的公司。想法很简单，就是用一个简单的 wiki 来包装一下 Ward Cunningham 的 FIT 工具，以编写验收测试（acceptance test）。

这件事情发生在 Maven 工具面世并“解决了”jar 文件问题之前。我当时坚信我们的产品不应该让用户下载超过一个的 jar 文件，我称这条规则为“下载即可执行”。这条规则指导了我们之后的很多决策。

第一个决策是根据 `FitNesse` 的需要专门编写了属于我们自己的 Web 服务器。这可能听起来很傻，即使在 2001 年，市面上也有足够多的开源的 Web 服务器可供选用。然而编写属于自己的 Web 服务器实际上是一个非常好的决策，因为一个只包含基本功能的 Web 服务器部署起来非常简单，它允许我们将任何与具体 Web 框架相关的决策延后[1]。

我们做的另一个早期决策是避免考虑数据库问题。我们当时确实考虑过使用 MySQL，但最后还是故意采用了一种与数据库无关的设计，而延后了这方面的决策。这部分的设计也很简单，就是在所有数据访问逻辑与数据仓库之间增加一个接口。

我们将数据访问方法放在一个名为 `WikiPage` 的接口中。这部分方法负责提供所需的查找、获取和保存页面的功能。当然，我们最初并没有具体实现这些方法，在开发不需要获取和保存数据的那部分功能时，我们只写了一个占位方法。

1 多年以后，我们成功地将 Velocity 框架插入了 `FitNesse` 之中。

确实，我们花了三个月时间来解决 wiki 文本与 HTML 之间的转换问题，这部分的工作与任何类型的数据存储都无关。所以我们在创建名为 MockWikiPage 的模块时，只写了一些空的数据访问方法。

最终，当这些占位方法不再支持我们所要开发的功能时，我们才需要真正实现数据访问。为此我们创建了一个名为 InMemoryPage 的 WikiPage 的派生类。在这个派生类中，实现了一系列数据访问方法来管理与 wiki 页面相关的哈希表（该哈希表会一直存在于内存中）。

就这样，我们一个接一个地开发这些业务功能，花了整整一年时间。事实上，FitNesse 程序能正常运行的第一个版本是这样的，它能让我们创建页面、链接其他页面、使用 wiki 格式完成所有的装饰，甚至用 FIT 工具运行测试。唯一不能做的就是真正地保存数据。

当该系统需要实现持久化时，我们又仔细考虑了一下是否采用 MySQL，最终还是觉得短期内没有必要，因为将哈希表写入文件是非常简单的。所以我们实现了一个 FileSystemWikiPage，用单一的大文件实现了这一功能，同时将我们的精力投入继续开发更多的新功能上。

三个月之后，我们得出一个结论，大文件的存储已经足够好了，完全没有必要使用 MySQL。就这样，我们不断推迟这个决策，到最后发现这个决策不需要做了。

如果不是我们的一个客户由于自己的需要而希望将数据存入 MySQL，我们的故事就到此为止了。事实上，在我们给他展示了 WikiPage 架构细节之后，他只用了一天时间就实现了一个能在 MySQL 上运行的系统，他所做的就是写了一个名为 MySQLWikiPage 的派生类，该系统就可以正常工作了。

我们曾经将这个实现和 FitNesse 一起打包分发，但是后来没有人真正用到过它，甚至连实现这个功能的用户最终也不再使用它了。

在开发 FitNesse 的早期，我们在业务逻辑和数据库之间画了一条边界线。这条线有效地防止了业务逻辑对数据库产生依赖，它只能访问简单的数据访问方法。这个决策使我们将与数据库选型和实现的决策推迟了超过一年。同时我们还能用文件系统进行实验，使我们最终换用了一个更好的解决方案。更重要的是，该架构在

需求真的出现时，没有阻止任何人采用 MySQL，甚至没有为其制造任何障碍。

在长达 18 个月的开发过程中，我们事实上没有用到过数据库，这意味着我们不需要面对表结构问题、查询问题、数据库服务器问题、密码问题、链接时间问题等一系列由于数据库带来的棘手问题。这样我们所有的测试都会进行得很快，因为它们都不依赖数据库。

简单来说，通过划清边界，我们可以推迟和延后一些细节性的决策，这最终会为我们节省大量的时间、避免大量的问题。这就是一个设计良好的架构所应该带来的助益。

应在何时、何处画这些线

边界线应该画在那些不相关的事情中间。GUI 与业务逻辑无关，所以两者之间应该有一条边界线。数据库与 GUI 无关，这两者之间也应该有一条边界线。数据库又与业务逻辑无关，所以两者之间也应该有一条边界线。

上面这些话，尤其是关于业务逻辑与数据库无关的部分可能会遭到部分读者的反对。大部分人都已经习惯性地认为数据库是与业务逻辑不可分割的了，有些人甚至认为，数据库相关逻辑部分本身就是业务逻辑的具体体现。

然而正如我们在第 18 章中将会讲到的，这个想法从根本上就是错误的。数据库应该是业务逻辑间接使用的一个工具。业务逻辑并不需要了解数据库的表结构、查询语言或其他任何数据库内部的实现细节。业务逻辑唯一需要知道的，就是有一组可以用来查询和保存数据的函数。这样一来，我们才可以将数据库隐藏在接口后面。

我们可以从图 17.1 中清晰地看到，`BusinessRules` 是通过 `DatabaseInterface` 来加载和保存数据的。而 `DatabaseAccess` 则负责实现该接口，以及其与实际 `Database` 的交互。

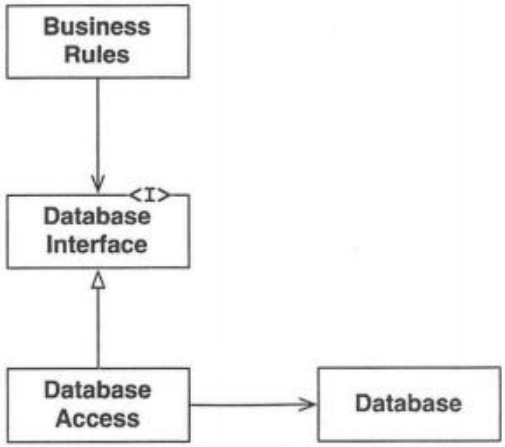


图 17.1：隐藏在接口背后的数据库

这里的类与接口仅仅是一个例子。在一个真实的应用程序中，将会有很多业务逻辑类、很多数据库接口类以及很多数据库访问的实现。不过，所有一切所遵循的模式应该是相似的。

那么这里的边界线应该被画在哪里？边界应该穿过继承关系，在 `DatabaseInterface` 之下（见图 17.2）。

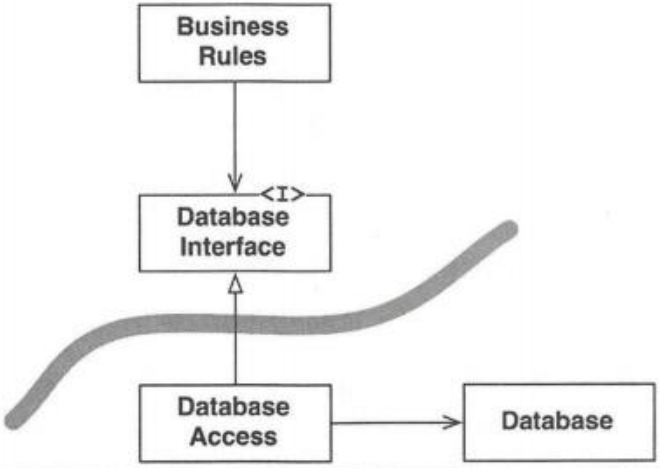


图 17.2：边界线

请注意，`DatabaseAccess` 类的那两个对外的箭头。这两个箭头都指向了远离

DatabaseAccess 类的方向，这意味着它们所指向的两个类都不知道 DatabaseAccess 类的存在。

下面让我们把抽象层次拉高一点，看一下包含多个业务逻辑类的组件与包含数据库及其访问类的组件之间是什么关系（见图 17.3）。

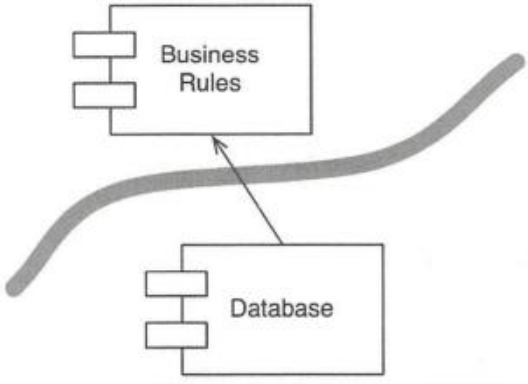


图 17.3：业务逻辑组件与数据库组件

请注意，图 17.3 中的箭头指向，它说明了 Database 组件知道 BusinessRules 组件的存在，而 BusinessRules 组件则不知道 Database 组件的存在。这意味着 DatabaseInterface 类是包含在 BusinessRules 组件中的，而 DatabaseAccess 类则被包含在 Database 组件中。

这个箭头的方向很重要。因为它意味着 Database 组件不会对 BusinessRules 组件形成干扰，但 Database 组件却不能脱离 BusinessRules 组件而存在。

如果读者对上面这段话感到困惑，请记住一点，Database 组件中包含了将 BusinessRules 组件中的函数调用转化为具体数据库查询语言的代码。这些转换代码当然必须知道 BusinessRules 组件的存在。

通过在这两个组件之间画边界线，并且让箭头指向 BusinessRules 组件，我们现在可以很容易地明白为什么 BusinessRules 组件可以使用任何一种数据库。在这里，Database 组件可以被替换为多种实现，BusinessRules 组件并不需要知道这件事。

数据库可以用 Oracle、MySQL、Couch 或者 Datomic，甚至大文件来实现。业务逻辑并不需要关心这件事。这意味着我们可以将与数据库相关的决策延后，先专注于编写业务逻辑的代码，进行测试，直到不得不选择数据库为止。

输入和输出怎么办

开发者和使用者经常会对系统边界究竟如何定义而感到困惑。由于 GUI 能够直观看到，就很自然地把 GUI 当成了系统本身。这些人以 GUI 的视角来定义整个系统，所以认为从系统开发一开始 GUI 部分就应该正常工作。这是错误的，这里他们没有意识到一个非常重要的原则，即 I/O 是无关紧要的。

这个原则可能一开始比较难以理解，毕竟我们经常从直觉上会以 I/O 的行为来定义系统的行为。以视频游戏为例，我们的主观体验是以界面反应为主的，这些反应来自屏幕、鼠标、按钮和声音等。但请不要忘了，这些界面背后存在着一个模型——一套非常复杂的数据结构和函数，那才是游戏真正的核心驱动力。更重要的是，该模型并不一定非要有一个界面。就算该游戏不显示在屏幕上，其模型也应该可以完成所有的任务逻辑，处理所有的游戏事件。因此，界面对模型——也就是业务逻辑来说——一点都不重要。

所以，`GUI` 和 `BusinessRules` 这两个组件之间也应该有一条边界线（见图 17.4）。再强调一次，在这里不重要的组件依赖于较为重要的组件，箭头指向的方向代表着组件之间的关系，`GUI` 关心 `BusinessRules`。

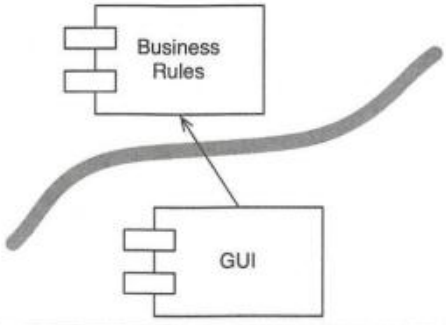


图 17.4：`GUI` 与 `BusinessRules` 之间的边界线

通过这条边界线以及这个箭头，我们可以看出 GUI 可以用任何一种其他形式的界面来代替[1]。BusinessRules 组件不需要了解这些细节。

插件式架构

综上所述，我们似乎可以基于数据库和 GUI 这两个为例来建立一种向系统添加其他组件的模式。这种模式与支持第三方插件的系统模式是一样的。

事实上，软件开发技术发展的历史就是一个如何想方设法方便地增加插件，从而构建一个可扩展、可维护的系统架构的故事。系统的核心业务逻辑必须和其他组件隔离，保持独立，而这些其他组件要么是可以去掉的，要么是有多种实现的（见图 17.5）。

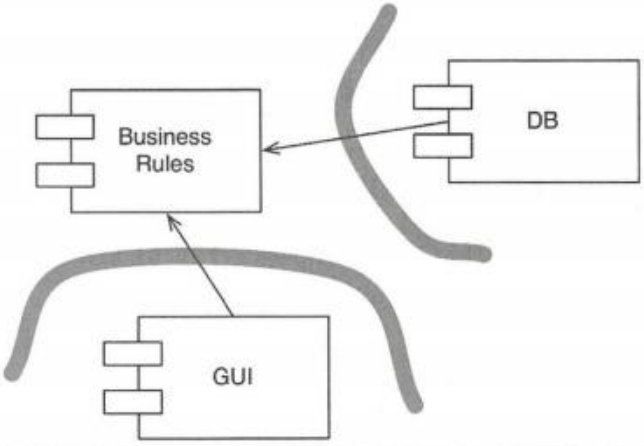


图 17.5：插件式架构

由于用户界面在这个设计中是以插件形式存在的，所以我们可以用插拔的方式切换很多不同类型的用户界面。可以是基于 Web 模式的、基于客户端/服务器端模式的、基于 SOA 模式的、基于命令行模式的或者基于其他任何类型的用户界面技术的。

数据库也类似。因为我们现在是将数据库作为插件来对待的，所以它就可以被

1 譬如，命令行界面（CUI）、Web 界面、移动端界面等。——编者注

替换成不同类型的 SQL 数据库、NoSQL 数据库，甚至基于文件系统的数据库，以及未来任何一种我们认为有必要发展的数据库技术。

当然，这些替换工作可能并不轻松，如果我们的系统一开始是按照 Web 方式部署的，那么为它写一个客户端/服务器端模型的 UI 插件就可能会比较困难一些。很可能业务逻辑与新 UI 之间的交互方式也要重新修改。但即使这样，插件式架构也至少为我们提供了这种实现的可能性。

插件式架构的好处

我们可以来看一下 ReSharper 和 Visual Studio 之间的关系。这两部分组件是由两个完全不同公司的人各自独立开发的，ReSharper 的开发者是 JetBrains 公司，它位于俄罗斯，而 Microsoft 公司则位于华盛顿州的雷德蒙市。很难想象有哪两个开发团队会比它们隔离得更彻底的了。

那么他们之间，是哪个团队可以影响另一个团队的工作呢？又是哪个团队可以对另一个团队的工作免疫呢？我们可以通过图 17.6 中的依赖关系来回答。很显然，是 ReSharper 的源代码依赖于 Visual Studio 的源代码。因此，是 ReSharper 团队无法干扰 Visual Studio 团队的工作，而 Visual Studio 团队却可以单方面中止 ReSharper 团队的任何工作。

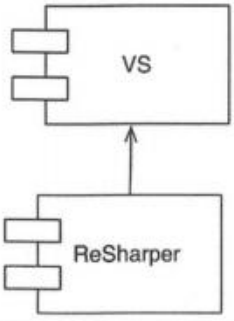


图 17.6：ReSharper 依赖于 Visual Studio

这是一种非常不对称的关系，但我们确实希望在自己的系统中构建这样的关系。因为这可以让部分组件对其他组件的变更免疫。例如，当有人修改 Web 页面格式或修改数据库表结构时，系统的业务逻辑部分就不应该受到影响。另外，我们也不希望系统中某一个部分发生的变更会导致其他不相关的部分出现问题，系统不应该这么脆弱。

将系统设计为插件式架构，就等于构建起了一面变更无法逾越的防火墙。换句话说，只要 GUI 是以插件形式插入系统的业务逻辑中的，那么 GUI 这边所发生的变更就不会影响系统的业务逻辑。

所以，边界线也应该沿着系统的变更轴来画。也就是说，位于边界线两侧的组件应该以不同原因、不同速率变化着。

一个系统的 GUI 与业务逻辑的变更原因、变更速率显然是不同的，所以二者中间应该有一条边界线。同样的，一个系统的业务逻辑与依赖注入框架之间的变更原因和变更速度也会不同，它们之间也应该画边界线。

这其实就是单一职责原则（SRP）的具体实现，SRP 的作用就是告诉我们应该在哪里画边界线。

本章小结

为了在软件架构中画边界线，我们需要先将系统分割成组件，其中一部分是系统的核心业务逻辑组件，而另一部分则是与核心业务逻辑无关但负责提供必要功能的插件。然后通过对源代码的修改，让这些非核心组件依赖于系统的核心业务逻辑组件。

其实，这也是一种对依赖反转原则（DIP）和稳定抽象原则（SAP）的具体应用，依赖箭头应该由底层具体实现细节指向高层抽象的方向。

第 18 章

边界剖析

一个系统的架构是由一系列软件组件以及它们之间的边界共同定义的。而这些边界有着多种不同的存在形式。在本章中，我们看看其中最常见的一些形式。

跨边界调用

在运行时，跨边界调用指的是边界线一侧的函数调用另一侧的函数，并同时传递数据的行为。构造合理的跨边界调用需要我们对源码中的依赖关系进行合理管控。

为什么需要管控源码中的依赖关系呢？因为当一个模块的源码发生变更时，其他模块的源码也可能会随之发生变更或重新编译，并需要重新部署。所谓划分边界，就是指在这些模块之间建立这种针对变更的防火墙。

令人生畏的单体结构

最简单、最常见的架构边界通常并没有一个固定的物理形式，它们只是对同一个进程、同一个地址空间内的函数和数据进行了某种划分。在第 16 章中，我们称之为源码层次上的解耦模式。

但是从部署的角度来看，这一切到最后都产生了一个单独的可执行文件——这就是所谓的单体结构。这个文件可能是一个静态链接形成的 C/C++项目，或是一个将一堆 Java 类绑定在一起的 jar 可执行文件，或是由一系列.NET 二进制文件组成的.EXE 文件等。

虽然这类系统的架构边界在部署过程中并不可见，但这并不意味着它们就不存在或者没有意义。因为即使最终所有的组件都被静态链接成了一个可执行文件，这些边界的划分对该系统各组件的独立开发也是非常有意义的。

因为这类架构一般都需要利用某种动态形式的多态[1]来管理其内部的依赖关系。

1 在 C++这样的编程语言中，静态形式的多态（例如泛型或者模板）有时候也是单体结构系统中的一种有效的依赖关系管理方式。然而，采用泛型这样的解耦模式并不能像动态形式的多态那样做到避免重新编译与重新部署。

这也是为什么面向对象编程近几十年来逐渐成为一种重要编程范式的原因之一。如果不采用面向对象编程模式或是类似的多态实现，架构师们就只能退回到用函数指针这种危险的模式来进行组件解耦的时代。由于大部分架构师认为大量采用函数指针过于危险，所以在那样的情况下，他们通常都在权衡利弊之后就干脆放弃划分组件了。

最简单的跨边界调用形式，是由低层客户端来调用高层服务函数，这种依赖关系在运行时和编译时会保持指向一致，都是从低层组件指向高层组件。

在图 18.1 中，我们可以看到控制流跨越边界的方向是从左向右的，Client 调用了 Service 上的函数 f()，并向它传递了一个 Data 实例。这里的<DS>标记是指 Data 是一个数据结构。Data 实例的具体传递方法可以是函数的调用参数，也可以是其他更复杂的传递方式。读者在这里需要注意的是，Data 的定义位于边界的被调用方一侧。

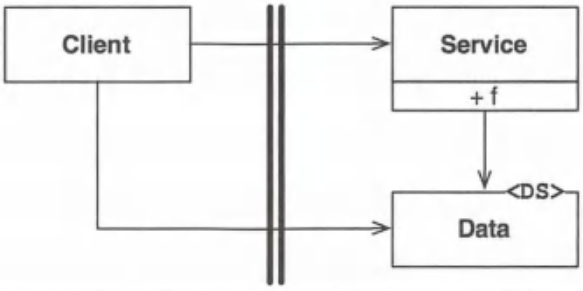


图 18.1：从低层组件跨越边界到达高层组件的控制流

但当高层组件中的客户端需要调用低层组件中的服务时，我们就需要运用动态形式的多态来反转依赖关系了。在这种情况下，系统在运行时的依赖关系与编译时的依赖关系就是相反的。

在图 18.2 中，控制流跨越边界的方向与之前是一样的，都是从左至右的。这里是高层组件 Client 通过 Service 接口调用了低层组件 ServiceImpl 上的函数 f()。但请读者注意，图 18.2 中所有的依赖关系却都是从右向左跨越边界的，方向是由低层组件指向高层组件的。同时，我们也应该注意到，这一次数据结构的定义是位于调用方这一侧的。

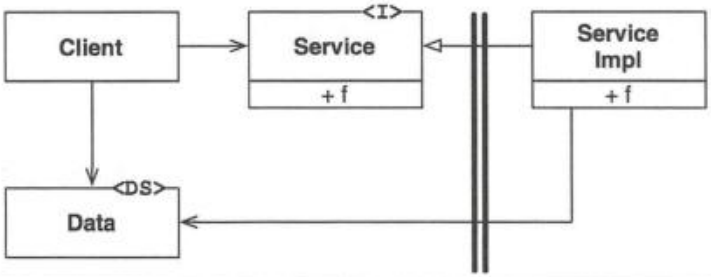


图 18.2：反方向跨越边界的控制流

即使是在一个单体部署、静态链接的可执行文件中，这种自律式的组件划分仍然可以极大地帮助整个项目的开发、测试与部署，使不同的团队可以独立开发不同的组件，不会互相干扰。高层组件与低层细节之间也可以得到良好的隔离，独立演进。

在单体结构中，组件之间的交互一般情况下都只是普通的函数调用，迅速而廉价，这就意味着这种跨源码层次解耦边界的通信会很频繁。

由于单体结构的部署需要编译所有源码，并且进行静态链接，这就意味着这些系统中的组件一般都会以源码形式交付。

部署层次的组件

下面我们来看看系统架构最常见的物理边界形式：动态链接库。这种形式包括.Net 的 DLL、Java 的 jar 文件、Ruby Gem 以及 UNIX 的共享库等。这种类型的组件在部署时不需要重新编译，因为它们都是以二进制形式或其他等价的可部署形式交付的。这里采用的就是部署层次上的解耦模式。部署这种类型的项目，就是将其所有可部署的单元打包成一个便于操作的文件格式，例如 WAR 文件，甚至可以只是一个目录（或者文件夹）。

除这一点以外，这种按部署层次解耦的组件与单体结构几乎是一样的，其所有的函数仍然处于同一个进程、同一个地址空间中。管理组件划分依赖关系的策略也

基本上是和上文一致的[1]。

与单体结构类似，按部署层次解耦的组件之间的跨边界调用也只是普通的函数调用，成本很低。虽然动态链接或运行时加载的过程本身可能会有一个一次性的调用成本，但它们之间的跨边界通信调用依然会很频繁。

线程

单体结构和按部署层次划分的组件都可以采用线程模型。当然，线程既不属于架构边界，也不属于部署单元，它们仅仅是一种管理并调度程序执行的方式。一个线程既可以被包含在单一组件中，也可以横跨多个组件。

本地进程

系统架构还有一个更明显的物理边界形式，那就是本地进程。本地进程一般是由命令行启动或其他等价的系统调用产生的。本地进程往往运行于单个处理器或多核系统的同一组处理器上，但它们拥有各自不同的地址空间。一般来说，现有的内存保护机制会使这些进程无法共享其内存，但它们通常可以用某种独立的内存区域来实现共享。

最常见的情况是，这些本地进程会用 socket 来实现彼此的通信。当然，它们也可以通过一些操作系统提供的方式来通信，例如共享邮件或消息队列。

每个本地进程都既可以是一个静态链接的单体结构，也可以是一个由动态链接组件组成的程序。在前一种情况下，若干个单体过程会被链接到同一个组件中。而在后一种情况下，这些单体过程可以共享同一个动态链接的可部署组件。

1　当然，静态形式的多态通常就不适用于这种情况了。

我们在这里可以将本地进程看成某种超级组件，该进程由一系列较低层次的组件组成，我们将通过动态形式的多态来管理它们之间的依赖关系。

另外，本地进程之间的隔离策略也与单体结构、二进制组件基本相同，其源码中的依赖关系跨越架构边界的方向是一致的，始终指向更高层次的组件。

对本地进程来说，这就意味着高层进程的源码中不应该包含低层进程的名字、物理内存地址或是注册表键名。请读者务必要记住，该系统架构的设计目标是让低层进程成为高层进程的一个插件。

本地进程之间的跨边界通信需要用到系统调用、数据的编码和解码，以及进程间的上下文切换，成本相对来说会更高一些，所以这里需要谨慎地控制通信的次数。

服务

系统架构中最强的边界形式就是服务。一个服务就是一个进程，它们通常由命令行环境或其他等价的系统调用来产生。服务并不依赖于具体的运行位置，两个互相通信的服务既可以处于单一物理处理器或多核系统的同一组处理器上，也可以彼此位于不同的处理器上。服务会始终假设它们之间的通信将全部通过网络进行。

服务之间的跨边界通信相对于函数调用来说，速度是非常缓慢的，其往返时间可以从几十毫秒到几秒不等。因此我们在划分架构边界时，一定要尽可能地控制通信次数。在这个层次上通信必须能够适应高延时情况。

除此之外，我们可以在服务层次上使用与本地进程相同的规则。也就是让较低层次服务成为较高层次服务的“插件”。为此，我们要确保高层服务的源码中没有包含任何与低层服务相关的物理信息（例如 URI）。

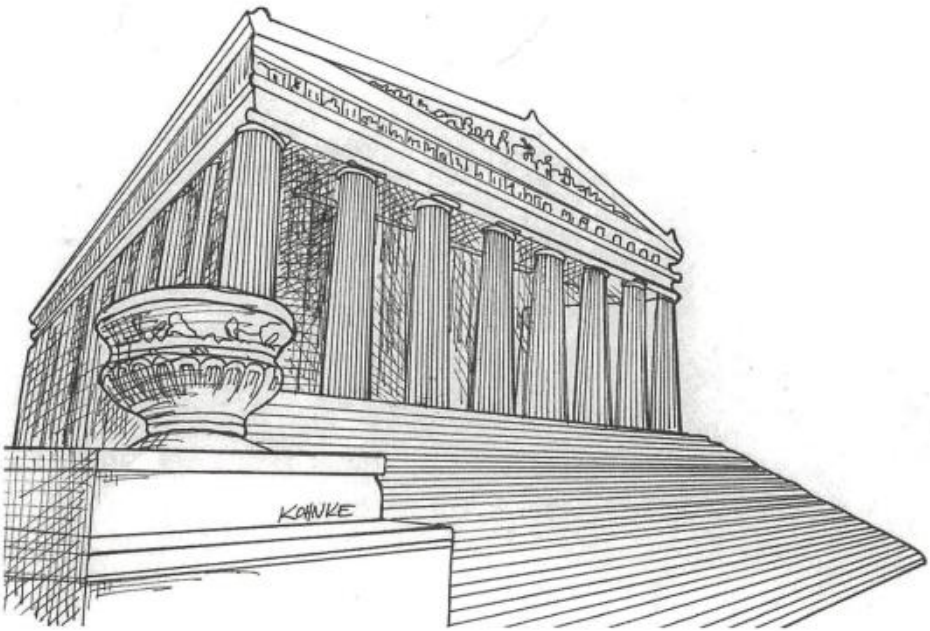
本章小结

除单体结构以外，大部分系统都会同时采用多种边界划分策略。一个按照服务层次划分边界的系统也可能会在某一部分采用本地进程的边界划分模式。事实上，服务经常不过就是一系列互相作用的本地进程的某种外在形式。无论是服务还是本地进程，它们几乎肯定都是由一个或多个源码组件组成的单体结构，或者一组动态链接的可部署组件。

这也意味着一个系统中通常会同时包含高通信量、低延迟的本地架构边界和低通信量、高延迟的服务边界。

第 19 章

策略与层次



本质上，所有的软件系统都是一组策略语句的集合。是的，可以说计算机程序不过就是一组仔细描述如何将输入转化为输出的策略语句的集合。

在大多数非小型系统（nontrivial system）中，整体业务策略通常都可以被拆解为多组更小的策略语句。一部分策略语句专门用于描述计算部分的业务逻辑，另一部分策略语句则负责描述计算报告的格式。除此之外，可能还会有一些用于描述如何校验输入数据的策略。

软件架构设计的工作重点之一就是，将这些策略彼此分离，然后将它们按照变更的方式进行重新分组。其中变更原因、时间和层次相同的策略应该被分到同一个组件中。反之，变更原因、时间和层次不同的策略则应该分属于不同的组件。

架构设计的工作常常需要将组件重排组合成为一个有向无环图。图中的每一个节点代表的是一个拥有相同层次策略的组件，每一条单向链接都代表了一种组件之间的依赖关系，它们将不同级别的组件链接起来。

这里提到的依赖关系是源码层次上的、编译期的依赖关系。这在 Java 语言中就是指 `import` 语句，在 C#语言中就是指 `using` 语句，在 Ruby 语言中就是指 `require` 语句。这里的依赖关系都是在编译过程中所必需的。

在一个设计良好的架构中，依赖关系的方向通常取决于它们所关联的组件层次。一般来说，低层组件被设计为依赖于高层组件。

层次（Level）

我们对“层次”是严格按照“输入与输出之间的距离”来定义的。也就是说，一条策略距离系统的输入/输出越远，它所属的层次就越高。而直接管理输入/输出的策略在系统中的层次是最低的。

在图 19.1 中，我们看到的是一个简单加密程序的数据流向图，该程序从输入设备读取字符，然后用查表法转换这些字符，并将转换后的字符输出到输出设备。我们将图中数据的流向用弯曲实心箭头标识了出来，而对于经精妙设计过的源码中的

依赖关系则使用直虚线来标识。

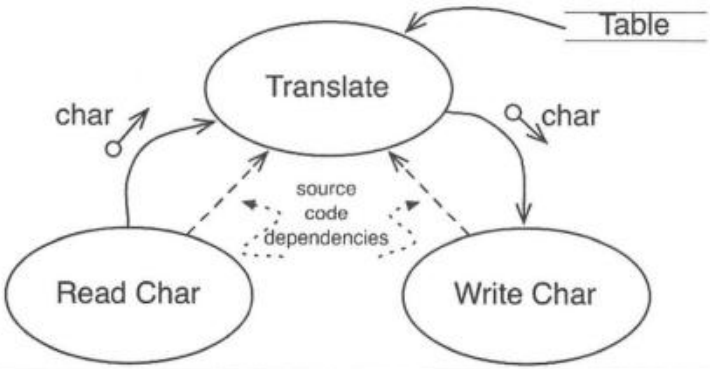


图 19.1：简单加密程序

在图 19.1 中，Translate 组件是这个系统中层次最高的组件，因为该组件距离系统输入/输出距离最远[1]。

另外需要注意的是，图 19.1 中的数据流向和源码中的依赖关系并不总处于同一方向上。这也是软件架构设计工作的一部分。我们希望源码中的依赖关系与其数据流向脱钩，而与组件所在的*层次挂钩*。

但我们很容易将这个加密程序写成下面这样，这就构成了一个不正确的架构：

```
function encrypt() {
  while(true)
    writeChar(translate(readChar()));
}
```

上面这个程序架构设计的错误在于，它让高层组件中的函数 encrypt() 依赖于低层组件中的函数 readChar() 与 writeChar()。

更好的系统架构设计应如图 19.2 所示。请注意图 19.2 中被虚线框起来的 Encrypt 类及其两个接口 CharReader 和 CharWriter。所有的依赖关系都指向了边界内部。这一切都说明它是该系统中最高层次的组件。

1 Meilir Page-Jones 在他的 *The Practical Guide to Structured Systems Design, 2nd Edition*（Yourdon Press 出版社，1988 年出版）一书中将这个组件称为“中央转换器”。

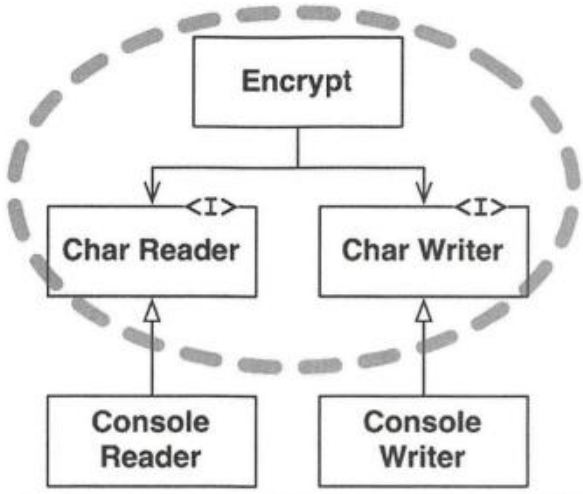


图 19.2：更好的系统架构设计图

在图 19.2 中，`ConsoleReader` 和 `ConsoleWriter` 都属于具体类。由于它们与输入/输出最近，因此属于低层组件。

另外应该注意的是，这个架构将高层的加密策略与低层的输入/输出策略解耦了。也就是说，当输入/输出部分的策略发生变更时，它们不太可能会影响加密部分的策略。

正如之前提到的，我们应该根据策略发生变更的方式来将它们分成不同的组件。变更原因和变更时间相同的策略应在 SRP 和 CCP 这两个原则的指导下合并为同一组件。离输入/输出最远的策略——高层策略——一般变更没有那么频繁。即使发生变更，其原因也比低层策略所在的组件更重大。反之，低层策略则很有可能会频繁地进行一些小变更。

例如，即使在这个简单的加密程序中，加密算法发生变更的可能性也要远小于 IO 设备发生变更的可能性。如果加密算法真的要变更，也很有可能比 I/O 设备的变更更重大。

通过将策略隔离，并让源码中的依赖方向都统一调整为指向高层策略，我们可以大幅度降低系统变更所带来的影响。因为一些针对系统低层组件的紧急小修改几乎不会影响系统中更高级、更重要的组件。

从另一个角度来说，低层组件应该成为高层组件的插件。图 19.3 中的组件图展示了这种关系，我们可以看到 Encryption 组件对 IODevices 组件的情况一无所知，而 IODevices 组件则依赖于 Encryption 组件。

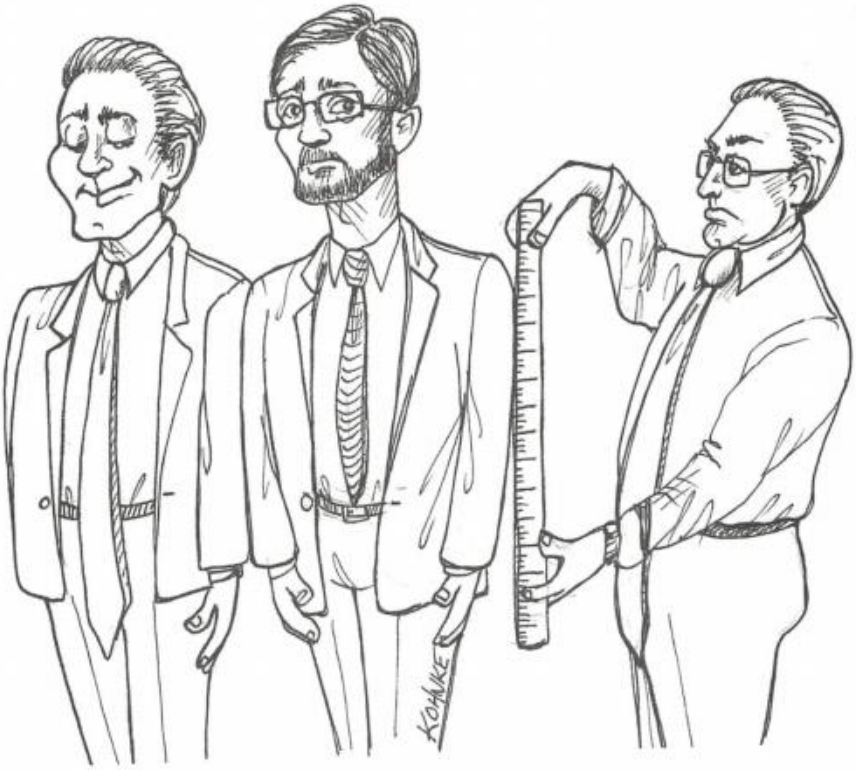
图 19.3：低层组件应该成为高层组件的插件

本章小结

综上所述，本章针对策略的讨论涉及单一职责原则（SRP）、开闭原则（OCP）、共同闭包原则（CCP）、依赖反转原则（DIP）、稳定依赖原则（SDP）以及稳定抽象原则（SAP）。读者可以自行结合之前的内容来匹配每个原则所适用的场景以及背后的原因。

第 20 章

业务逻辑



如果我们要将自己的应用程序划分为业务逻辑和插件两部分，就必须更仔细地了解业务逻辑究竟是什么，它到底有几种类型。

严格地讲，业务逻辑就是程序中那些真正用于赚钱或省钱的业务逻辑与过程。更严格地讲，无论这些业务逻辑是在计算机上实现的，还是人工执行的，它们在省钱/赚钱上的作用都是一样的。

例如，银行要对借贷收取 *N*%利息这个逻辑就是银行获取收入方面的一条业务逻辑。对它来说，我们通过计算机来计算利息，还是让一个银行职员用计算器来计算利息并不重要。

我们通常称这些逻辑为“关键业务逻辑”，因为它们是一项业务的关键部分，不管有没有自动化系统来执行这项业务，这一点是不会改变的。

“关键业务逻辑”通常会需要处理一些数据，例如，在借贷的业务逻辑中，我们需要知道借贷的数量、利率以及还款日程。

我们将这些数据称为“关键业务数据”，这是因为这些数据无论自动化程序存在与否，都必须要存在。

关键业务逻辑和关键业务数据是紧密相关的，所以它们很适合被放在同一个对象中处理。我们将这种对象称为“业务实体（*Entity*）”[1]。

业务实体

业务实体实际上就是计算机系统中的一种对象，这种对象中包含了一系列用于操作关键数据的业务逻辑。这些实体对象要么直接包含关键业务数据，要么可以很容易地访问这些数据。业务实体的接口层则是由那些实现关键业务逻辑、操作关键业务数据的函数组成的。

例如，在图 20.1 中，我们看到的是一个对应于借贷业务的实体类 Loan 的 UML

1 这个概念的名字是 Ivar Jacobson 起的（请参见 I.Jacobson 等人合著的 *Object Oriented Software Engineering* 一书，Addison-Wesley 出版社，1992 年出版）。

图。如你所见，该类中包含了三个关键业务数据，以及三个代表了其关键业务逻辑的接口。

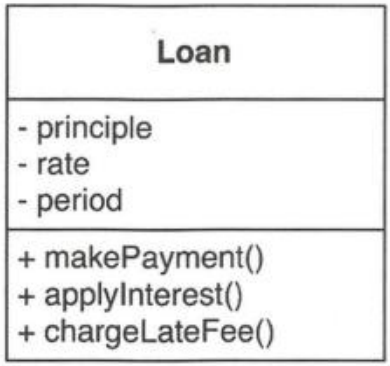


图 20.1：对应于借贷业务的实体类 Loan 的 UML 图

当我们创建这样一个类时，其实就是在将软件中具体实现了该关键业务的部分聚合在一起，将其与自动化系统中我们所构建的其他部分隔离区分。这个类独自代表了整个业务逻辑，它与数据库、用户界面、第三方框架等内容无关。该类可以在任何一个系统中提供与其业务逻辑相关的服务，它不会去管这个系统是如何呈现给用户的，数据是如何存储的，或者是以何种方式运行的。总而言之，业务实体这个概念中应该只有业务逻辑，没有别的。

有些读者可能会担心我在这里把业务实体解释成一个类。不是这样的，业务实体不一定非要用面向对象编程语言的类来实现。业务实体这个概念只要求我们将关键业务数据和关键业务逻辑绑定在一个独立的软件模块内。

用例

并不是所有的业务逻辑都是一个纯粹的业务实体。例如，有些业务逻辑是通过定义或限制自动化系统的运行方式来实现赚钱或省钱的业务的。这些业务逻辑就不能靠人工来执行，它们只有在作为自动化系统的一部分时才有意义。

例如，假设我们现在有一个银行职员们用来新建借贷的应用程序，银行可能设

计的业务逻辑是，银行职员必须首先收集、验证客户的联系信息，确保客户的信用值在 500 以上，然后才允许向用户提供借贷还款的预估值。因此，银行就必须要求在设计其计算机系统时确保两件事：首先，客户必须能通过屏幕填写所有的联系信息并且让其通过相关验证；其次，客户只有在其信用值大于既定阈值时才能进入还款预估页。

我们在上面所描述的就是一个用例（*use case*）[1]。用例本质上就是关于如何操作一个自动化系统的描述，它定义了用户需要提供的输入数据、用户应该得到的输出信息以及产生输出所应该采取的处理步骤。当然，用例所描述的是某种特定应用情景下的业务逻辑，它并非业务实体中所包含的关键业务逻辑。

下面，让我们来看看图 20.2 中的这个用例。请注意，图 20.2 的最后一行中提到的“客户”，这里的客户是指代表客户的业务实体，其中包含了处理银行与客户之间关系的关键业务逻辑。

为新贷款收集联系信息

输入：名字、地址、生日、驾照号码或社会安全号码等。
输出：显示所输入的信息以供确认，并且包括信用值。

主要步骤：
1. **接受并且校验名字。**
2. **校验地址、生日、驾照号码、社会安全号码等。**
3. **获取客户的信用值。**
4. **如果信用值小于500，则跳转到拒绝页。**
5. **否则，创建客户对象，并且跳转到还款预估页。**

图 20.2：用例示范

如上所示，用例中包含了对如何调用业务实体中的关键业务逻辑的定义。简而言之，用例控制着业务实体之间的交互方式。

除此之外，这里还应该注意，用例除非正式地描述了数据流入/流出接口以外，

1 我们之前已经介绍过了。

并不详细描述用户界面。也就是说，如果我们只看用例，是没有办法分辨出系统是在 Web 平台上交付的，还是交付了某种富客户端；或者是以命令行模式交付的，还是以一个内部服务模式交付的。

这是非常重要的。用例并不描述系统与用户之间的接口，它只描述该应用在某些特定情景下的业务逻辑，这些业务逻辑所规范的是用户与业务实体之间的交互方式，它与数据流入/流出系统的方式无关。

在我们的系统中，用例本身也是一个对象，该对象中包含了一个或多个实现了特定应用情景的业务逻辑函数。当然除此之外，用例对象中也包含了输入数据、输出数据以及相关业务实体的引用，以方便调用。

当然，业务实体并不会知道是哪个业务用例在控制它们，这也是依赖反转原则（DIP）的另一个应用情景。也就是像业务实体这样的高层概念是无须了解像用例这样的低层概念的。反之，低层的业务用例却需要了解高层的业务实体。

那么，为什么业务实体属于高层概念，而用例属于低层概念呢？因为用例描述的是一个特定的应用情景，这样一来，用例必然会更靠近系统的输入和输出。而业务实体是一个可以适用于多个应用情景的一般化概念，相对地离系统的输入和输出更远。所以，用例依赖于业务实体，而业务实体并不依赖于用例。

请求和响应模型

在通常情况下，用例会接收输入数据，并产生输出数据。但在一个设计良好的架构中，用例对象通常不应该知道数据展现给用户或者其他组件的方式。很显然，我们当然不会希望这些用例类中的代码出现 HTML 和 SQL。

因此，用例类所接收的输入应该是一个简单的请求性数据结构，而返回输出的应该是一个简单的响应性数据结构。这些数据结构中不应该存在任何依赖关系，它们并不派生自 `HttpRequest` 和 `HttpResponse` 这样的标准框架接口。这些数据接口应该与 Web 无关，也不应该了解任何有关用户界面的细节。

这种独立性非常关键，如果这里的请求和响应模型不是完全独立的，那么用到这些模型的用例就会依赖于这些模型所带来的各种依赖关系。

可能有些读者会选择直接在数据结构中使用对业务实体对象的引用。毕竟，业务实体与请求/响应模型之间有很多相同的数据。但请一定不要这样做！这两个对象存在的意义是非常、非常不一样的。随着时间的推移，这两个对象会以不同的原因、不同的速率发生变更。所以将它们以任何方式整合在一起都是对共同闭包原则（CCP）和单一职责原则（SRP）的违反。这样做的后果，往往会导致代码中出现很多分支判断语句和中间数据。

本章小结

业务逻辑是一个软件系统存在的意义，它们属于核心功能，是系统用来赚钱或省钱的那部分代码，是整个系统中的皇冠明珠。

这些业务逻辑应该保持纯净，不要掺杂用户界面或者所使用的数据库相关的东西。在理想情况下，这部分代表业务逻辑的代码应该是整个系统的核心，其他低层概念的实现应该以插件形式接入系统中。业务逻辑应该是系统中最独立、复用性最高的代码。

第 21 章

尖叫的软件架构

假设我们现在正在查看某个建筑的设计架构图，那么在这个反映建筑设计师精心设计成果的文件中，究竟应该包括怎样的架构图呢？

如果这是一幅单户住宅的建筑架构图，那么我们很可能会先看到一个大门，然后是一条连接到起居室的通道，同时可能还会看到一个餐厅。接着，距离餐厅不远处应该会有一个厨房，可能厨房附近还会有一个非正式用餐区，或一个亲子房。当我们阅读这个架构图时，应该不会怀疑这是一个单户住宅。几乎整个建筑设计都在尖叫着告诉你：这是一个“家”。

假设我们阅读的是一幅图书馆的建筑设计图，情况也差不多。我们应该会先看到一个超大入口，然后是一个用于签到/签出的办公区，接下来是阅读区、小型会议室，以及一排排的书架区。同样，几乎整个建筑设计都在尖叫着跟你说：这是一个“图书馆”。

那么，我们的应用程序的架构设计又会“喊”些什么呢？当我们查看它的顶层结构目录，以及顶层软件包中的源代码时，它们究竟是在喊“健康管理系统”“账务系统”“库存管理系统”，还是在喊：“Rails”“Spring/Hibernate”“ASP”这样的技术名词呢？

架构设计的主题

在这里，再次推荐读者仔细阅读 Ivar Jacobson 关于软件架构设计的那本书：*Object Oriented Software Engineering*，请读者注意这本书的副标题：*A Use Case Driven Approach*（业务用例驱动的设计方式）。在这本书中，Jacobson 提出了一个观点：软件的系统架构应该为该系统的用例提供支持。这就像住宅和图书馆的建筑计划满篇都在非常明显地凸显这些建筑的用例一样，软件系统的架构设计图也应该非常明确地凸显该应用程序会有哪些用例。

架构设计不是（或者说不应该是）与框架相关的，这件事不应该是基于框架来完成的。对于我们来说，框架只是一个可用的工具和手段，而不是一个架构所规范的内容。如果我们的架构是基于框架来设计的，它就不能基于我们的用例来设计了。

架构设计的核心目标

一个良好的架构设计应该围绕着用例来展开，这样的架构设计可以在脱离框架、工具以及使用环境的情况下完整地描述用例。这就好像一个住宅建筑设计的首要目标应该是满足住宅的使用需求，而不是确保一定要用砖来构建这个房子。架构师应该花费很多精力来确保该架构的设计在满足用例需要的情况下，尽可能地允许用户能自由地选择建筑材料（砖头、石料或者木材）。

而且，良好的架构设计应该尽可能地允许用户推迟和延后决定采用什么框架、数据库、Web 服务以及其他与环境相关的工具。框架应该是一个可选项，良好的架构设计应该允许用户在项目后期再决定是否采用 Rails、Spring、Hibernate、Tomcat、MySQL 这些工具。同时，良好的架构设计还应该让我们很容易改变这些决定。总之，良好的架构设计应该只关注用例，并能将它们与其他的周边因素隔离。

那 Web 呢

Web 究竟是不是一种架构？如果我们的系统需要以 Web 形式来交付，这是否意味着我们只能采用某种系统架构？当然不是！Web 只是一种交付手段——一种 IO 设备——这就是它在应用程序的架构设计中的角色。换句话说，应用程序采用 Web 方式来交付只是一个实现细节，这不应该主导整个项目的结构设计。事实上，关于一个应用程序是否应该以 Web 形式来交付这件事，它本身就应该是一个被推迟和延后的决策。一个系统应该尽量保持它与交付方式之间的无关性。在不更改基础架构设计的情况下，我们应该可以将一个应用程序交付成命令行程序、Web 程序、富客户端程序、Web 服务程序等任何一种形式的程序。

框架是工具而不是生活信条

当然，框架通常可以是非常强大、非常有用的。但框架作者往往对自己写出的框架有着极深的信念，他们所写出来的使用手册一般都是从如何成为该框架的虔诚

信徒的角度来描绘如何使用这个框架的。甚至这些框架的使用者所写的教程也会出现这种传教士模式。他们会告诉你某个框架是能包揽一切、超越一切、解决一切问题的存在。

这不应该成为你的观点。

我们一定要带着怀疑的态度审视每一个框架。是的，采用框架可能会很有帮助，但采用它们的成本呢？我们一定要懂得权衡如何使用一个框架，如何保护自己。无论如何，我们需要仔细考虑如何能保持对系统用例的关注，避免让框架主导我们的架构设计。

可测试的架构设计

如果系统架构的所有设计都是围绕着用例来展开的，并且在使用框架的问题上保持谨慎的态度，那么我们就应该可以在不依赖任何框架的情况下针对这些用例进行单元测试。另外，我们在运行测试的时候不应该运行 Web 服务，也不应该需要连接数据库。我们测试的应该只是一个简单的业务实体对象，没有任何与框架、数据库相关的依赖关系。总而言之，我们应该通过用例对象来调度业务实体对象，确保所有的测试都不需要依赖框架。

本章小结

一个系统的架构应该着重于展示系统本身的设计，而并非该系统所使用的框架。如果我们要构建的是一个医疗系统，新来的程序员第一次看到其源码时就应该知道这是一个医疗系统。新来的程序员应该先了解该系统的用例，而非系统的交付方式。他们可能会走过来问你：

“我看到了一些看起来像是模型的代码——但它们的视图和控制器在哪里？”

这时你的回答应该是：

“哦，我们现在先不考虑这些细节问题，回头再来决定应该怎么做。”

第 22 章

整洁架构

在过去的几十年中，我们曾见证过一系列关于系统架构的想法被提出，列举如下。

- 六边形架构（Hexagonal Architecture）（也称为端口与适配器架构，Ports and Adpaters）：该架构由 Alistair Cockburn 首先提出。Steve Freeman 和 Nat Pryce 在他们合写的著作 *Growing Object Oriented Software with Tests* 一书中对该架构做了隆重的推荐。
- DCI 架构：由 James Coplien 和 Trygve Reenskaug 首先提出。
- BCE 架构：由 Ivar Jacobson 在他的 *Object Oriented Software Engineering: A Use-Case Driven Approach* 一书中首先提出。

虽然这些架构在细节上各有不同，但总体来说是非常相似的。它们都具有同一个设计目标：按照不同关注点对软件进行切割。也就是说，这些架构都会将软件切割成不同的层，至少有一层是只包含该软件的业务逻辑的，而用户接口、系统接口则属于其他层。

按照这些架构设计出来的系统，通常都具有以下特点。

- 独立于框架：这些系统的架构并不依赖某个功能丰富的框架之中的某个函数。框架可以被当成工具来使用，但不需要让系统来适应框架。
- 可被测试：这些系统的业务逻辑可以脱离 UI、数据库、Web 服务以及其他的外部元素来进行测试。
- 独立于 UI：这些系统的 UI 变更起来很容易，不需要修改其他的系统部分。例如，我们可以在不修改业务逻辑的前提下将一个系统的 UI 由 Web 界面替换成命令行界面。
- 独立于数据库：我们可以轻易将这些系统使用的 Oracle、SQL Server 替换成 Mongo、BigTable、CouchDB 之类的数据库。因为业务逻辑与数据库之间已经完成了解耦。
- 独立于任何外部机构：这些系统的业务逻辑并不需要知道任何其他外部接口的存在。

下面我们要通过图 22.1 将上述所有架构的设计理念综合成为一个独立的理念。

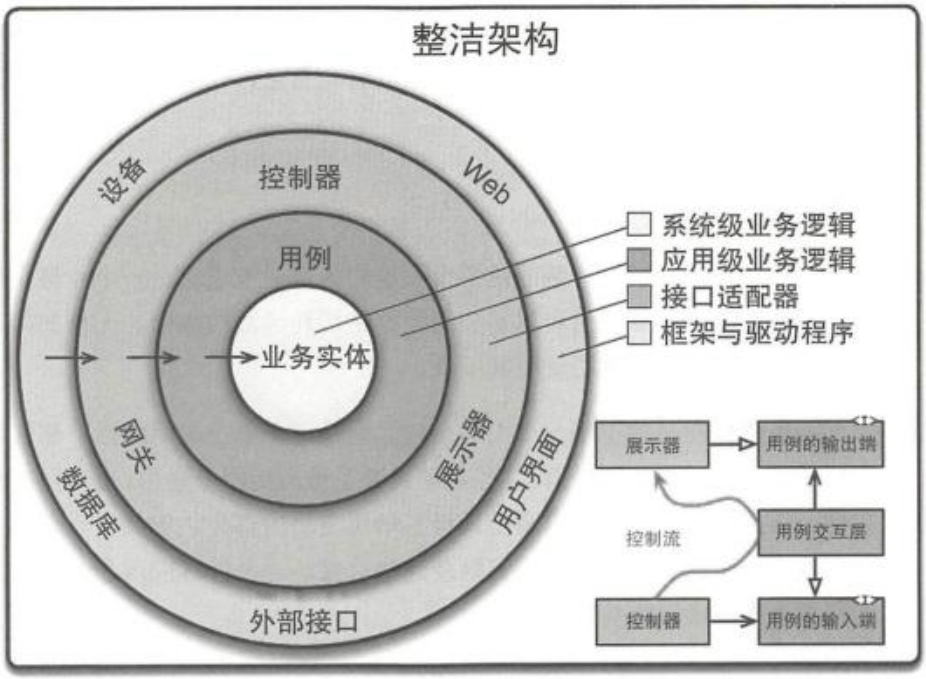


图 22.1：整洁架构

依赖关系规则

图 22.1 中的同心圆分别代表了软件系统中的不同层次，通常越靠近中心，其所在的软件层次就越高。基本上，外层圆代表的是机制，内层圆代表的是策略。

当然这其中有一条贯穿整个架构设计的规则，即它的依赖关系规则：

> 源码中的依赖关系必须只指向同心圆的内层，即由低层机制指向高层策略。

换句话说，就是任何属于内层圆中的代码都不应该牵涉外层圆中的代码，尤其是内层圆中的代码不应该引用外层圆中代码所声明的名字，包括函数、类、变量以及一切其他有命名的软件实体。

同样的道理，外层圆中使用的数据格式也不应该被内层圆中的代码所使用，尤其是当数据格式是由外层圆的框架所生成时。总之，我们不应该让外层圆中发生的任何变更影响到内层圆的代码。

业务实体

业务实体这一层中封装的是整个系统的关键业务逻辑，一个业务实体既可以是一个带有方法的对象，也可以是一组数据结构和函数的集合。无论如何，只要它能被系统中的其他不同应用复用就可以。

如果我们在写的不是一个大型系统，而是一个单一应用的话，那么我们的业务实体就是该应用的业务对象。这些对象封装了该应用中最通用、最高层的业务逻辑，它们应该属于系统中最不容易受外界影响而变动的部分。例如，一个针对页面导航方式或者安全问题的修改不应该触及这些对象，一个针对应用在运行时的行为所做的变更也不应该影响业务实体。

用例

软件的用例层中通常包含的是特定应用场景下的业务逻辑，这里面封装并实现了整个系统的所有用例。这些用例引导了数据在业务实体之间的流入/流出，并指挥着业务实体利用其中的关键业务逻辑来实现用例的设计目标。

我们既不希望在这一层所发生的变更影响业务实体，同时也不希望这一层受外部因素（譬如数据库、UI、常见框架）的影响。用例层应该与它们都保持隔离。

然而，我们知道应用行为的变化会影响用例本身，因此一定会影响用例层的代码。因为如果一个用例的细节发生了变化，这一层中的某些代码自然要受到影响。

接口适配器

软件的接口适配器层中通常是一组数据转换器，它们负责将数据从对用例和业务实体而言最方便操作的格式，转化成外部系统（譬如数据库以及 Web）最方便操作的格式。例如，这一层中应该包含整个 GUI MVC 框架。展示器、视图、控制器

都应该属于接口适配器层。而模型部分则应该由控制器传递给用例，再由用例传回展示器和视图。

同样的，这一层的代码也会负责将数据从对业务实体与用例而言最方便操作的格式，转化为对所采用的持久性框架（譬如数据库）最方便的格式。总之，在从该层再往内的同心圆中，其代码就不应该依赖任何数据库了。譬如说，如果我们采用的是 SQL 数据库，那么所有的 SQL 语句都应该被限制在这一层的代码中——而且是仅限于那些需要操作数据库的代码。

当然，这一层的代码也会负责将来自外部服务的数据转换成系统内用例与业务实体所需的格式。

框架与驱动程序

图 22.1 中最外层的模型层一般是由工具、数据库、Web 框架等组成的。在这一层中，我们通常只需要编写一些与内层沟通的黏合性代码。

框架与驱动程序层中包含了所有的实现细节。Web 是一个实现细节，数据库也是一个实现细节。我们将这些细节放在最外层，这样它们就很难影响到其他层了。

只有四层吗

图 22.1 中所显示的同心圆只是为了说明架构的结构，真正的架构很可能会超过四层。并没有某个规则约定一个系统的架构有且只能有四层。然而，这其中的依赖关系原则是不变的。也就是说，源码层面的依赖关系一定要指向同心圆的内侧。层次越往内，其抽象和策略的层次越高，同时软件的抽象程度就越高，其包含的高层策略就越多。最内层的圆中包含的是最通用、最高层的策略，最外层的圆包含的是最具体的实现细节。

跨越边界

在图 22.1 的右下侧，我们示范的是在架构中跨边界的情况。具体来说就是控制器、展示器与下一层的用例之间的通信过程。请注意这里控制流的方向：它从控制

器开始，穿过用例，最后执行展示器的代码。但同时我们也该注意到，源码中的依赖方向却都是向内指向用例的。

这里，我们通常采用依赖反转原则（DIP）来解决这种相反性。例如，在 Java 这一类的语言中，可以通过调整代码中的接口和继承关系，利用源码中的依赖关系来限制控制流只能在正确的地方跨越架构边界。

假设某些用例代码需要调用展示器，这里一定不能直接调用，因为这样做会违反依赖关系原则：内层圆中的代码不能引用其外层的声明。我们需要让业务逻辑代码调用一个内层接口（图 22.1 中的“用例输出端”），并让展示器来负责实现这个接口。

我们可以采用这种方式跨越系统中所有的架构边界。利用动态多态技术，我们将源码中的依赖关系与控制流的方向进行反转。不管控制流原本的方向如何，我们都可以让它遵守架构的依赖关系规则。

哪些数据会跨越边界

一般来说，会跨越边界的数据在数据结构上都是很简单的。如果可以的话，我们会尽量采用一些基本的结构体或简单的可传输数据对象。或者直接通过函数调用的参数来传递数据。另外，我们也可以将数据放入哈希表，或整合成某种对象。这里最重要的是这个跨边界传输的对象应该有一个独立、简单的数据结构。总之，不要投机取巧地直接传递业务实体或数据库记录对象。同时，这些传递的数据结构中也不应该存在违反依赖规则的依赖关系。

例如，很多数据库框架会返回一个便于查询的结果对象，我们称之为“行结构体”。这个结构体不应该跨边界向架构的内层传递。因为这等于让内层的代码引用外层代码，违反依赖规则。

因此，当我们进行跨边界传输时，一定要采用内层最方便使用的形式。

一个常见的应用场景

接下来，我们将会在图 22.2 中看到一个基于 Web 的、使用数据库的 Java 系统。在该系统中，Web 服务器会从用户那里收集信息，并将它们交给左上角的 Controller。然后，Controller 将这些信息数据包装成一个简单的 Java 对象，并让该对象穿越 InputBoundary 被传递到 UseCaseInteractor。接下来，我们会让 UseCaseInteractor 解析数据，并通过它来控制与 Entities 的交互。同时，我们还会用 DataAccessInterface 将 Entities 需要用到的数据从 Database 加载到内存中。随后，UseCaseInteractor 会负责从 Entities 收集数据，并将 OutputData 组装成另一个简单的 Java 对象。最后，OutputData 会穿越 OutputBoundary 被传递给 Presenter。

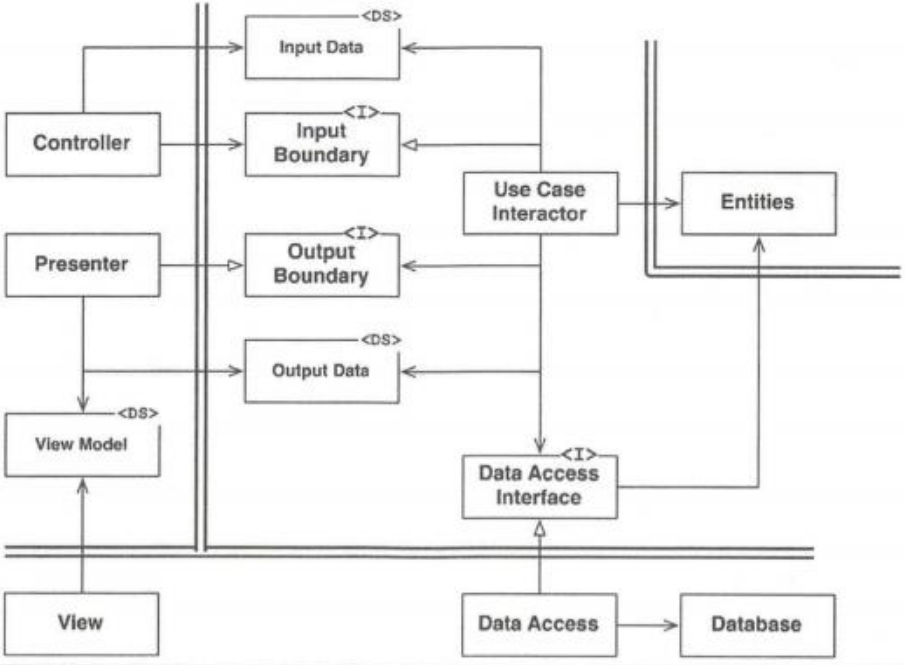


图 22.2：一个基于 Web 的、使用数据库的常见 Java 程序

接下来，Presenter 的任务是将 OutputData 重新打包成可展示的 ViewModel，这也是一个简单的 Java 对象。ViewModel 中基本上只包含字符串和一些 View 都会用到的开关数据。同时，OutputData 中可能会包含一些 Date 对象，Presenter 会将其格式化成可对用户展示的字符串，并将其填充到 ViewModel 中。同理，Currency 对象和其他业务相关的数据也会经历类似的操作。如你所见，Button 和 MenuItems 的命名定义位于 ViewModel 中，并且其中还包括了用于告知 View 层 Button 和 MenuItems 是否可用的开关数据。

我们可以看出，View 除了将 ViewModel 中的数据转换成 HTML 格式之外，并没有其他功能。

最后，读者必须注意一下这里的依赖关系方向。所有跨边界的依赖线都是指向内的，这很好地遵守了架构的依赖关系规则。

本章小结

如你所见，遵守上面这些简单的规范并不困难，这样做能在未来避免各种令人头疼的问题。通过将系统划分层次，并确保这些层次遵守依赖关系规则，就可以构建出一个天生可测试的系统，这其中的好处是不言而喻的。而且，当系统外层的这些数据库或 Web 框架过时的时候，我们还可以很轻松地替换它们。

第 23 章

展示器和谦卑对象

在第 22 章中，我们引入了展示器（presenter）的概念，展示器实际上是采用谦卑对象（*humble object*）模式的一种形式，这种设计模式可以很好地帮助识别和保护系统架构的边界。事实上，第 22 章所介绍的整洁架构中就充满了大量谦卑对象的实现体。

谦卑对象模式

谦卑对象模式[1]最初的设计目的是帮助单元测试的编写者区分容易测试的行为与难以测试的行为，并将它们隔离。其设计思路非常简单，就是将这两类行为拆分成两组模块或类。其中一组模块被称为谦卑（Humble）组，包含了系统中所有难以测试的行为，而这些行为已经被简化到不能再简化了。另一组模块则包含了所有不属于谦卑对象的行为。

例如，GUI 通常是很难进行单元测试的，因为让计算机自行检视屏幕内容，并检查指定元素是否出现是非常难的事情。然而，GUI 中的大部分行为实际上是很容易被测试的。这时候，我们就可以利用谦卑对象模式将 GUI 的这两种行为拆分成展示器与视图两部分。

展示器与视图

视图部分属于难以测试的谦卑对象。这种对象的代码通常应该越简单越好，它只应负责将数据填充到 GUI 上，而不应该对数据进行任何处理。

展示器则是可测试的对象。展示器的工作是负责从应用程序中接收数据，然后按视图的需要将这些数据格式化，以便视图将其呈现在屏幕上。例如，如果应用程序需要在屏幕上展示一个日期，那么它传递给展示器的应该是一个 `Date` 对象。然后展示器会将该对象格式化成所需的字符串形式，并将其填充到视图模型中。

1 请参考 Gerard Meszaros 所著的 *xUnit Test Patterns: Refactoring Test Code* 一书第 695 页（Addison-Wesley 出版社，2007 年出版）。

如果应用程序需要在屏幕上展示金额，那么它应该将 Currency 对象传递给展示器。展示器随后会将这个对象按所需的小数位数进行格式化，并加上对应的货币标识符，形成一个字符串存放在视图模型中。如果需要将负数金额显示成红色，那么该视图模型中就应该有一个简单的布尔值被恰当地设置。

另外，应用程序在屏幕上的每个按钮都应该有其对应的名称，这些名称也是由展示器在视图模型中设置的。如果某个按钮需要变灰，展示器就应该将相应的开关变量设置成对应的布尔值。同样，菜单中每个菜单项所显示的值，也应该是一个个由展示器加载到视图模型中的字符串。应用程序在屏幕上显示的每个单选项、多选项以及文本框的名字也都如此，在视图模型中都有相应的字符串和布尔值可供展示器做对应的设置。即使屏幕上要加载的是一个数值表，展示器也应该负责把这些数值格式化成具有表格属性的字符串，以供视图使用。

总而言之，应用程序所能控制的、要在屏幕上显示的一切东西，都应该在视图模型中以字符串、布尔值或枚举值的形式存在。然后，视图部分除了加载视图模型所需要的值，不应该再做任何其他事情。因此，我们才能说视图是谦卑对象。[1]

测试与架构

众所周知，强大的可测试性是一个架构的设计是否优秀的显著衡量标准之一。谦卑对象模式就是这方面的一个非常好的例子。我们将系统行为分割成可测试和不可测试两部分的过程常常就也定义了系统的架构边界。展示器与视图之间的边界只是多种架构边界中的一种，另外还有许多其他边界。

1　“谦卑”在这里是拟人化的，指难以测试的对象清晰地认识到自己的局限性，只发挥自己的桥梁和通信作用，并不从中干预信息的传输。——译者注

数据库网关

对于用例交互器(interactor)与数据库中间的组件，我们通常称之为数据库网关[1]。这些数据库网关本身是一个多态接口，包含了应用程序在数据库上所要执行的创建、读取、更新、删除等所有操作。例如，如果应用程序需要知道所有昨天登录系统的用户的姓，那么 `UserGateway` 接口就应该包含一个 `getLastNamesOfUsersWhoLoggedInAfter` 方法，接收一个 `Date` 参数，并返回一个包含姓的列表。

另外，我们之前说过，SQL 不应该出现在用例层的代码中，所以这部分的功能就需要通过网关接口来提供，而这些接口的实现则要由数据库层的类来负责。显然，这些实现也应该都属于谦卑对象，它们应该只利用 SQL 或其他数据库提供的接口来访问所需要的数据。与之相反，交互器则不属于谦卑对象，因为它们封装的是特定应用场景下的业务逻辑。不过，交互器尽管不属于谦卑对象，却是可测试的，因为数据库网关通常可以被替换成对应的测试桩和测试替身类。

数据映射器

让我们继续数据库方面的话题，现在我们来思考一下 Hibernate 这类的 ORM 框架应该属于系统架构中的哪一层呢？

首先，我们要弄清楚一件事：对象关系映射器（ORM）事实上是压根就不存在的。道理很简单，对象不是数据结构。至少从用户的角度来说，对象内部的数据应该都是私有的，不可见的，用户在通常情况下只能看到对象的公有函数。因此从用户角度来说，对象是一些操作的集合，而不是简单的数据结构体。

与之相反，数据结构体则是一组公开的数据变量，其中不包含任何行为信息。所以 ORM 更应该被称为“数据映射器”，因为它们只是将数据从关系型数据库加

1 请参考 Martin Fowler 等人所著的 *Patterns of Enterprise Application Architecture* 一书（Addison-Wesley 出版社，2003 年出版）第 466 页。

载到了对应的数据结构中。

那么，这样的 ORM 系统应该属于系统架构中的哪一层呢？当然是数据库层。ORM 其实就是在数据库和数据库网关接口之间构建了另一种谦卑对象的边界。

服务监听器

如果我们的应用程序需要与其他服务进行某种交互，或者该应用本身要提供某一套服务，我们在相关服务的边界处也会看到谦卑对象模式吗？

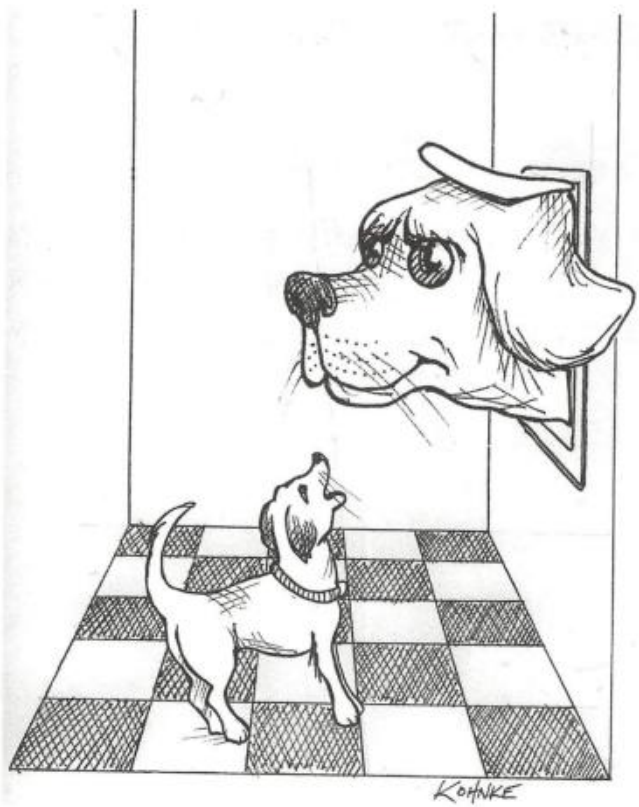
答案是肯定的。我们的应用程序会将数据加载到简单的数据结构中，并将这些数据结构跨边界传输给那些能够将其格式化并传递其他外部服务的模块。在输入端，服务监听器会负责从服务接口中接收数据，并将其格式化成该应用程序易用的格式。总而言之，上述数据结构可以进行跨服务边界的传输。

本章小结

在每个系统架构的边界处，都有可能发现谦卑对象模式的存在。因为跨边界的通信肯定需要用到某种简单的数据结构，而边界会自然而然地将系统分割成难以测试的部分与容易测试的部分，所以通过在系统的边界处运用谦卑对象模式，我们可以大幅地提高整个系统的可测试性。

第 24 章

不完全边界



构建完整的架构边界是一件很耗费成本的事。在这个过程中，需要为系统设计双向的多态边界接口，用于输入和输出的数据结构，以及所有相关的依赖关系管理，以便将系统分割成可独立编译与部署的组件。这里会涉及大量的前期工作，以及大量的后期维护工作。

在很多情况下，一位优秀的架构师都会认为设计架构边界的成本太高了——但为了应对将来可能的需要，通常还是希望预留一个边界。

但这种预防性设计在敏捷社区里是饱受诟病的，因为它显然违背了 YAGNI 原则（“You Aren't Going to Need It”，意即“不要预测未来的需要”）。然而，架构师的工作本身就是要做这样的预见性设计，这时候，我们就需要引入不完全边界（partial boundary）的概念了。

省掉最后一步

构建不完全边界的一种方式就是在将系统分割成一系列可以独立编译、独立部署的组件之后，再把它们构建成一个组件。换句话说，在将系统中所有的接口、用于输入/输出的数据格式等每一件事都设置好之后，仍选择将它们统一编译和部署为一个组件。

显然，这种不完全边界所需要的代码量以及设计的工作量，和设计完整边界时是完全一样的。但它省去了多组件管理这部分的工作，这就等于省去了版本号管理和发布管理方面的工作——这其中的工作量其实可不小。

这也是 FitNesse 项目早期所采取的策略。我们在设计 Web 服务器之初就将它设计为一个可以独立于 wiki 和测试部分的组件。该设计背后的想法是我们未来可能需要使用该 Web 组件来构建其他应用程序。但是同时，我们又不想让用户下载两个组件。正如我们之前所说，该项目其中的一个设计目标是实现让用户下载即可运行。我们希望用户只需下载一个 jar 文件就立即可以执行，不需要再去寻找其他的 jar 文件，更不需要操心版本兼容性等问题。

在这里，FitNesse 项目的故事也可以作为一个反例来说明这种设计的危险性。随着时间的推移，我们慢慢发现，将 Web 组件独立的需求越来越少，Wiki 组件与 Web 组件的隔离也弱化了。到如今，如果真想要再分离 Web 组件的话，会需要不少工作量。

单向边界

在设计一套完整的系统架构边界时，往往需要用反向接口来维护边界两侧组件的隔离性。而且，维护这种双向的隔离性，通常不会是一次性的工作，它需要我们持续地长期投入资源维护下去。

在图 24.1 中，你会看到一个临时占位的，将来可被替换成完整架构边界的更简单的结构。这个结构采用了传统的策略模式（*strategy pattern*）。如你所见，其 `Client` 使用的是一个由 `ServiceImpl` 类实现的 `ServiceBoundary` 接口。

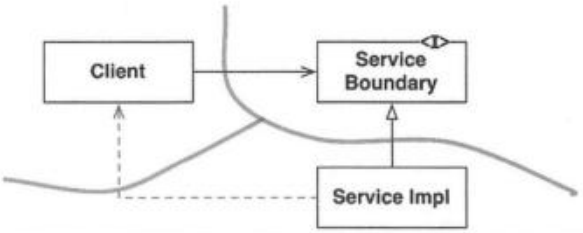


图 24.1：策略模式

很明显，上述设计为未来构建完整的系统架构边界打下了坚实基础。为了未来将 `Client` 与 `ServiceImpl` 隔离，必要的依赖反转已经做完了。同时，我们也能清楚地看到，图中的虚线箭头代表了未来有可能很快就会出现的隔离问题。由于没有采用双向反向接口，这部分就只能依赖开发者和架构师的自律性来保证组件持久隔离了。

门户模式

下面，我们再来看一个更简单的架构边界设计：采用门户模式（*facade pattern*），其架构如图 24.2 所示。在这种模式下，我们连依赖反转的工作都可以省了。这里的边界将只能由 `Facade` 类来定义，这个类的背后是一份包含了所有服务函数的列表，它会负责将 `Client` 的调用传递给对 `Client` 不可见的服务函数。

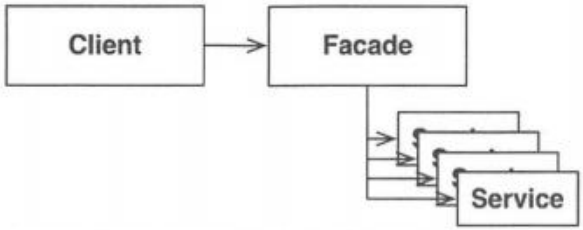


图 24.2：门户模式

但需要注意的是，在该设计中，`Client` 会传递性地依赖于所有的 `Service` 类。在静态类型语言中，这就意味着对 `Service` 类的源码所做的任何修改都会导致 `Client` 的重新编译。另外，我们应该也能想象得到为这种结构建立反向通道是多容易的事。

本章小结

在本章，我们介绍了三种不完全地实现架构边界的简单方法。当然，这类边界还有许多种其他实现方式，本章所介绍的这三种策略只为示范之用。

每种实现方式都有相应的成本和收益。每种方式都有自己所适用的场景，它们可以被用来充当最终完整架构边界的临时替代品。同时，如果这些边界最终被证明是没有必要存在的，那么也可以被自然降解。

架构师的职责之一就是预判未来哪里有可能会需要设置架构边界，并决定应该以完全形式还是不完全形式来实现它们。

第 25 章

层次与边界

人们通常习惯于将系统分成三个组件：UI、业务逻辑和数据库。对于一些简单系统来说，的确可以这样，但稍复杂一些系统的组件就远不止三个了。

以一个简单的计算机游戏为例。粗略看来，它好像也很符合三个组件的架构设定。首先，让 UI 负责接收用户输入的数据，并将数据传递给游戏的业务逻辑。然后，游戏的业务逻辑会将游戏状态保存在某种持久化数据结构中。但是，仅仅是这样而已吗？

基于文本的冒险游戏：*Hunt The Wumpus*

现在让我们往上面的设想中加入一些细节。假设这个游戏是 1972 年风靡一时的基于文本的冒险游戏：*Hunt the Wumpus*。这个游戏的操作是通过一些像 GO EAST 和 SHOOT WEST 这样的简单文字命令来完成的。玩家在输入命令之后，计算机就会返回玩家角色所看到的、闻到的、听到的或体会到的事情。在这个游戏中，玩家会在一系列洞穴中追捕 Wumpus[1]。玩家必须避开陷阱、陷坑以及其他一系列危险。如果有兴趣，在网上很容易找到该游戏的规则说明。

现在，假设我们决定保留这种基于文本的 UI，但是需要将 UI 与游戏业务逻辑之间的耦合解开，以便我们的游戏版本可以在不同地区使用不同的语言。也就是说，游戏的业务逻辑与 UI 之间应该用一种与自然语言无关的 API 来进行通信，而由 UI 负责将 API 传递进来的信息转换成合适的自然语言。

如果我们能管理好源码中的依赖关系，就应该像图 25.1 所展示的那样，多个 UI 组件复用同一套游戏业务逻辑。而游戏的业务逻辑组件不知道，也不必知道 UI 正在使用哪一种自然语言。

1　一种虚构的神秘怪兽。——译者注

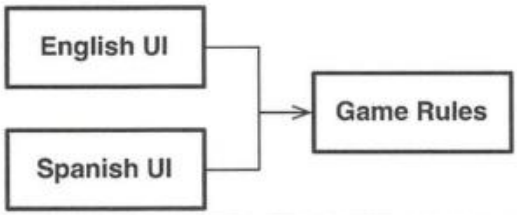


图 25.1：多个 UI 组件复用同一个游戏业务逻辑组件

同时，假设玩家在游戏中的状态会被保存在某种持久化存储介质中——有可能闪存，也有可能是某种云端存储，或只是本机内存。无论怎样，我们都并不希望游戏引擎了解这些细节。所以，我们仍然需要创建一个 API 来负责游戏的业务逻辑组件与数据存储组件之间的通信。

由于我们不会希望让游戏的业务逻辑依赖于不同种类的数据存储，所以这里的设计也要合理地遵守依赖关系原则，这样的话，该游戏的结构应如图 25.2 所示。

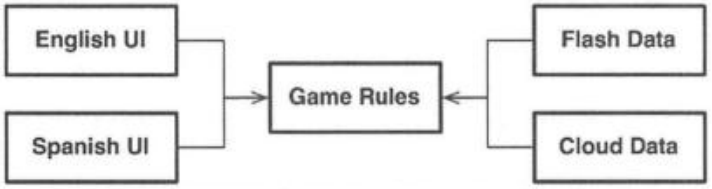


图 25.2：遵循依赖关系规则的设计

可否采用整洁架构

很显然，这里具备了采用整洁架构方法所需要的一切，包括用例、业务实体以及对应的数据结构都有了[1]，但我们是否已经找到了所有相应的架构边界呢？

例如，语言并不是 UI 变更的唯一方向。我们可能还会需要变更文字输入/输出的方式。例如，我们的输入/输出可以采用命令行窗口，或者用短信息，或者采用某种聊天程序。这里的可能性有很多。

1 当然，同样明显的是，这种小游戏实际上无须采用整洁架构。毕竟，整个程序差不多只需要 200 行代码。这里只是用这个简单程序来示范如何为系统找出相应的架构边界。

这就意味着这类变更应该有一个对应的架构边界。也许我们需要构造一个 API，以便将语言部分与通信部分隔开，这样一来，该设计的结构应如图 25.3 所示。

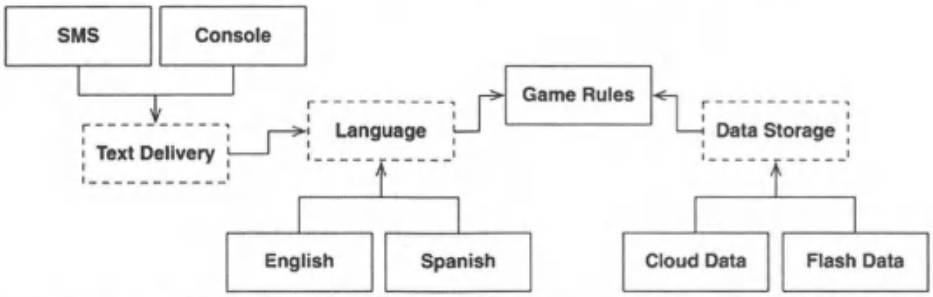


图 25.3：修正后的设计图

在图 25.3 中可以看到，现在系统的结构已经变得有点复杂了。在该图中，虚线框代表的是抽象组件，它们所定义的 API 通常要交由其上下层的组件来实现。例如 Language 部分的 API 是由 English 和 Spanish 这两个组件来实现的。

我们也可以看到 GameRules 与 Language 这两个组件之间的交互是通过一个由 GameRules 定义，并由 Language 实现的 API 来完成的。同样的，Language 与 TextDelievery 之间的交互也是通过由 Language 定义，并由 TextDelievery 实现的 API 来完成。这些 API 的定义和维护都是由使用方来负责的，而非实现方。

如果我们进一步查看 GameRules 内部，就会发现 GameRules 组件的代码中使用的 Boundary 多态接口是由 Language 组件来实现的；同时还会发现 Language 组件使用的 Boundary 多态接口由 GameRules 代码实现。

如果再探究一下 Language 组件，我们也会看到类似的情况：它的 Boundary 多态接口是在 TextDelievery 组件的代码中实现的，而 TextDelievery 使用的 Boundary 多态接口则由 Language 来实现。

在所有这些场景中，由 Boundary 接口所定义的 API 都是由其使用者的上一层组件负责维护的。

不同的具体实现类，例如 English、SMS、CloudData 都实现了由抽象的 API

组件所定义的多态接口。例如，Language 组件中定义的多态接口是由 English 和 Spanish 这两个组件来定义的。

我们可以去掉所有的具体实现类，只保留 API 组件来进一步简化上面这张设计图，其简化的结果如图 25.4 所示。

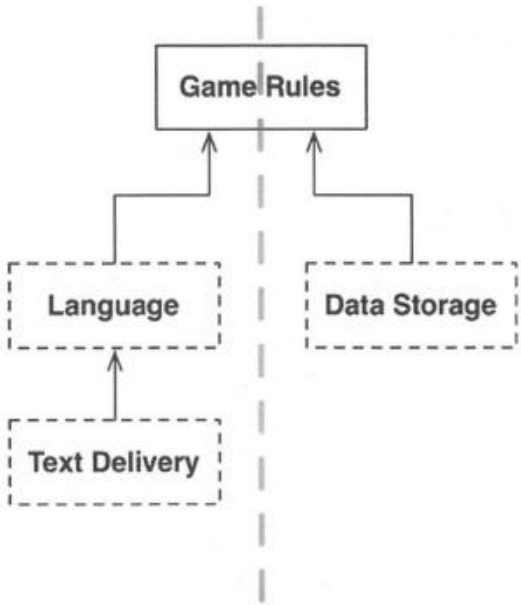


图 25.4：简化版设计图

请注意图 25.4 中的朝向设计，所有的箭头都是朝上的。这样 GameRules 组件就被放在顶层的位置。这种朝向设计很好地反映了 GameRules 作为最高层策略组件的事实。

下面，我们来考虑一下信息流的方向。首先，所有来自用户的信息都会通过左下角的 TextDelievery 组件传入。当这些信息被上传到 Language 组件时，就会转换为具体的命令输入给 GameRules 组件。然后，GameRules 组件会负责处理用户的输入，并将数据发送给右下角的 DataStorage 组件。

接下来，GameRules 会将输出向下传递到 Language 组件，将其转成合适的语言并通过 TextDelievery 将该语言传递给用户。

这种设计方式将数据流分成两路[1]。左侧的数据流关注如何与用户通信，而右侧的数据流关注的是数据持久化。两条数据流在顶部的 GameRules 汇聚[2]。GameRules 组件是所有数据的最终处理者。

交汇数据流

那么，这个例子中是否永远只有这两条数据流呢？当然不是。假设我们现在要在网络上与多个其他玩家一起玩这个游戏，就会需要一个网络组件，如图 25.5 所示。这样一来，我们有了三条数据流，它们都由 GameRules 组件所控制。

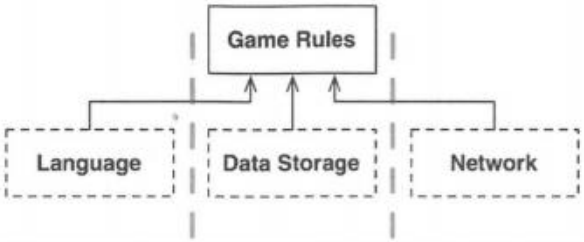


图 25.5：增加一个网络组件

由此可见，随着系统的复杂化，组件在架构中自然会分裂出多条数据流来。

数据流的分割

此时你可能会认为所有的数据流最终都会汇聚到一个组件上。生活要是果真如此简单，那就真是太好了！现实情况往往不如人所愿啊。

我们可以再来看一下 *Hunt The Wumpu* 这个游戏的 GameRules 组件。游戏的部分业务逻辑处理的是玩家在地图中的行走。这一部分需要知道游戏中的洞穴如何相

1　如果你觉得箭头的方向有问题，请记住这里箭头的方向代表的是源码中的依赖关系，不是数据流的方向。

2　很久以前，我们曾称顶端的组件为中央变换器（central transform），关于这方面的详细信息请参考 *Practical Guide to Structured Systems Design*（第二版，Meilir Page-Jones 出版社，1988 年）。

连，每个洞穴中有什么物体存在，还要知道如何将玩家从一个洞穴移到另一个洞穴，以及如何触发各种需要玩家处理的事件。

但是，游戏中还有一组更高层次的策略——这些策略负责了解玩家的血量，以及每个事件的后果和影响。这些策略既可以让玩家逐渐损失血量，也可能由于发现食物而增加血量。总而言之，游戏的低层策略会负责向高层策略传递事件，例如 `FoundFood` 和 `FellInPit`。而高层组件则要管理玩家状态（如图 25.6 所示），最终该策略将会决定玩家在游戏中的输赢。

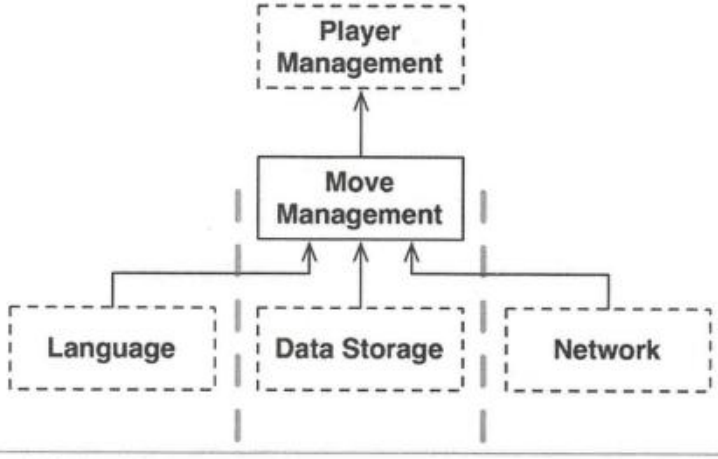


图 25.6：管理玩家的高层策略

这些究竟是否属于架构边界呢？是否需要设计一个 API 来分割 `MoveManagement` 和 `PlayerManagement` 呢？在回答这些问题之前，让我们把问题弄得更有意思一点，再往里面加上微服务吧！

假设我们现在面对的是一个可以面向海量玩家的新版 *Hunt The Wumpus* 游戏。它的 `MoveManagmenet` 组合是由玩家的本地计算机来处理的。而 `PlayerManagement` 组件则由服务端来处理。但 `PlayerMangament` 组件会为所有连接上它的 `MoveManagement` 组件提供一个微服务的 API。

在图 25.7 中，我们为该游戏绘制了一个简化版的设计图。现实中的 `Network` 组件通常会比图中的更复杂一些——但这里的已经足够说明情况了。在图中，可以看到 `MoveMangament` 与 `PlayerManagment` 之间存在一个完整的系统架构边界。

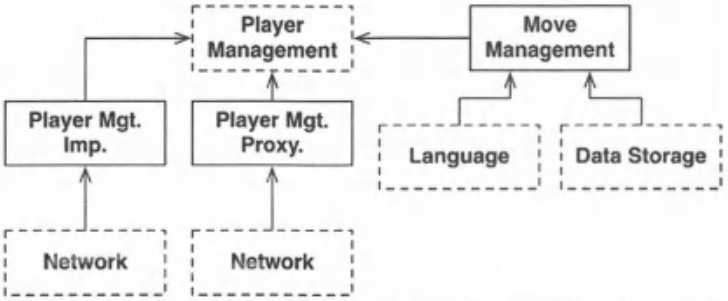


图 25.7：添加一个微服务的 API

本章小结

本章究竟想讨论什么呢？为什么要将一个极为简单的、在 Kornshell 中只需 200 行代码就能写完的小程序扩展成具有这些系统架构边界的复杂程序？

我们设计这个例子的目的就是为了证明架构边界可以存在于任何地方。作为架构师，我们必须要小心审视究竟在什么地方才需要设计架构边界。另外，我们还必须弄清楚完全实现这些边界将会带来多大的成本。

同时，我们也必须要了解如果事先忽略了这些边界，后续再添加会有多么困难——哪怕有覆盖广泛的测试，严加小心的重构也于事无补。

所以作为架构师，我们应该怎么办？这个问题恐怕没有答案。一方面，就像一些很聪明的人多年来一直告诉我们的那样，不应该将未来的需求抽象化。这就是 YAGNI 原则：“You aren’t going to need it”，臆想中的需求事实上往往是不存在的。这是一句饱含智慧的建议，因为过度的工程设计往往比工程设计不足还要糟糕。但另一方面，如果我们发现自己在某个位置确实需要设置一个架构边界，却又没有事先准备的时候，再添加边界所需要的成本和风险往往是很高的。

现实就是这样。作为软件架构师，我们必须有一点未卜先知的能力。有时候要依靠猜测——当然还要用点脑子。软件架构师必须仔细权衡成本，决定哪里需要设计架构边界，以及这些地方需要的是完整的边界，还是不完全的边界，还是可以忽略的边界。

而且，这不是一次性的决定。我们不能在项目开始时就决定好哪里需要设计边界，哪里不需要。相反，架构师必须持续观察系统的演进，时刻注意哪里可能需要设计边界，然后仔细观察这些地方会由于不存在边界而出现哪些问题。

当出现问题时，我们还需要权衡一下实现这个边界的成本，并拿它与不实现这个边界的成本对比——这种对比经常需要反复地进行。我们的目标是找到设置边界的优势超过其成本的拐点，那就是实现该边界的最佳时机。

持之以恒，一刻也不能放松。

第 26 章

Main 组件

在所有的系统中，都至少要有一个组件来负责创建、协调、监督其他组件的运转。我们将其称为 Main 组件。

最细节化的部分

Main 组件是系统中最细节化的部分——也就是底层的策略，它是整个系统的初始点。在整个系统中，除了操作系统不会再有其他组件依赖于它了。Main 组件的任务是创建所有的工厂类、策略类以及其他的全局设施，并最终将系统的控制权转交给最高抽象层的代码来处理。

Main 组件中的依赖关系通常应该由依赖注入框架来注入。在该框架将依赖关系注入到 Main 组件之后，Main 组件就应该可以在不依赖于该框架的情况下自行分配这些依赖关系了。

请记住，Main 组件是整个系统中细节信息最多的组件。

下面，我们来看一下最新版 *Hunt the Wumpus* 游戏的 Main 组件。请注意这里加载字符串的方法，这些字符串全都是我们不想让游戏主体代码了解的内容。

```
public class Main implements HtwMessageReceiver {
  private static HuntTheWumpus game;
  private static int hitPoints = 10;
  private static final List<String> caverns = new ArrayList<>();
  private static final String[] environments = new String[]{
    "bright",
    "humid",
    "dry",
    "creepy",
    "ugly",
    "foggy",
    "hot",
    "cold",
    "drafty",
    "dreadful"
  };
```

```
private static final String[] shapes = new String[] {
  "round",
  "square",
  "oval",
  "irregular",
  "long",
  "craggy",
  "rough",
  "tall",
  "narrow"
};

private static final String[] cavernTypes = new String[] {
  "cavern",
  "room",
  "chamber",
  "catacomb",
  "crevasse",
  "cell",
  "tunnel",
  "passageway",
  "hall",
  "expanse"
};

private static final String[] adornments = new String[] {
  "smelling of sulfur",
  "with engravings on the walls",
  "with a bumpy floor",
  "",
  "littered with garbage",
  "spattered with guano",
  "with piles of Wumpus droppings",
  "with bones scattered around",
  "with a corpse on the floor",
  "that seems to vibrate",
  "that feels stuffy",
  "that fills you with dread"
 };
```

接下来是main函数。请注意这里是如何使用HtwFactory来构建这个游戏的。我们可以看到这里传入了一个名为htw.game.HuntTheWumpusFacade的类。由于这个类中的细节信息比Main组件还多，变更也更频繁，因此这样做可以避免这个类的变更导致Main组件的重新编译和重新部署。

```
public static void main(String[] args) throws IOException {
    game = HtwFactory.makeGame("htw.game.HuntTheWumpusFacade",
                                    new Main());
    createMap();
    BufferedReader br = new BufferedReader(new InputStreamReader (System.in));
      game.makeRestCommand().execute();
      while (true) {
        System.out.println(game.getPlayerCavern());
        System.out.println("Health: " + hitPoints + " arrows: " +
                                game.getQuiver());
        HuntTheWumpus.Command c = game.makeRestCommand();
        System.out.println(">");
        String command = br.readLine();
        if (command.equalsIgnoreCase("e"))
          c = game.makeMoveCommand(EAST);
        else if (command.equalsIgnoreCase("w"))
          c = game.makeMoveCommand(WEST);
        else if (command.equalsIgnoreCase("n"))
          c = game.makeMoveCommand(NORTH);
        else if (command.equalsIgnoreCase("s"))
          c = game.makeMoveCommand(SOUTH);
        else if (command.equalsIgnoreCase("r"))
          c = game.makeRestCommand();
        else if (command.equalsIgnoreCase("sw"))
          c = game.makeShootCommand(WEST);
        else if (command.equalsIgnoreCase("se"))
          c = game.makeShootCommand(EAST);
        else if (command.equalsIgnoreCase("sn"))
          c = game.makeShootCommand(NORTH);
        else if (command.equalsIgnoreCase("ss"))
          c = game.makeShootCommand(SOUTH);

        else if (command.equalsIgnoreCase("q"))
```

```
            return;

          c.execute();
        }
    }
```

你还应该注意到 main 函数中创建了输入数据流，并纳入了游戏的主循环。主循环将负责处理简单的输入指令，但它会将具体的处理过程交给其他更高层次的组件来处理。

最后，Main 组件还要负责生成整个游戏的地图。

```
  private static void createMap() {
    int nCaverns = (int) (Math.random() * 30.0 + 10.0);
    while (nCaverns-- > 0)
      caverns.add(makeName());

    for (String cavern : caverns) {
      maybeConnectCavern(cavern, NORTH);
      maybeConnectCavern(cavern, SOUTH);
      maybeConnectCavern(cavern, EAST);
      maybeConnectCavern(cavern, WEST);
    }

    String playerCavern = anyCavern();
    game.setPlayerCavern(playerCavern);
    game.setWumpusCavern(anyOther(playerCavern));
    game.addBatCavern(anyOther(playerCavern));
    game.addBatCavern(anyOther(playerCavern));
    game.addBatCavern(anyOther(playerCavern));

    game.addPitCavern(anyOther(playerCavern));
    game.addPitCavern(anyOther(playerCavern));
    game.addPitCavern(anyOther(playerCavern));

    game.setQuiver(5);
  }

  // 此处删除了大量代码……
}
```

我们在这里的重点是要说明 Main 组件是整个系统中的一个底层模块，它处于整洁架构的最外圈，主要负责为系统加载所有必要的信息，然后再将控制权转交回系统的高层组件。

本章小结

Main 组件也可以被视为应用程序的一个插件——这个插件负责设置起始状态、配置信息、加载外部资源，最后将控制权转交给应用程序的其他高层组件。另外，由于 Main 组件能以插件形式存在于系统中，因此我们可以为一个系统设计多个 Main 组件，让它们各自对应于不同的配置。

例如，我们既可以设计专门针对开发环境的 Main 组件，也可以设计专门针对测试的或者生产环境的 Main 组件。除此之外，我们还可以针对要部署的国家、地区甚至客户设计不同的 Main 组件。

当我们将 Main 组件视为一种插件时，用架构边界将它与系统其他部分隔离开这件事，在系统的配置上是不是就变得更容易了呢？

第 27 章

服务：宏观与微观

面向服务的“架构”以及微服务“架构”近年来非常流行，其中的原因如下：

- 服务之间似乎是强隔离的，但是下文我们会讲到，并不完全是这样。
- 服务被认为是支持独立开发和部署的，同样，下文我们也会讲到，并不完全是这样。

面向服务的架构

首先，我们来批判“只要使用了服务，就等于有了一套架构”这种思想。这显然是完全错误的。如前文所述，架构设计的任务就是找到高层策略与低层细节之间的架构边界，同时保证这些边界遵守依赖关系规则。所谓的服务本身只是一种比函数调用方式成本稍高的，分割应用程序行为的一种形式，与系统架构无关。

当然，这里并不是说所有的服务都应该具有系统架构上的意义。有时候，用服务这种形式来隔离不同平台或进程中的程序行为这件事本身就很重要——不管它们是否遵守依赖关系规则。我们只是认为，服务本身并不能完全代表系统架构。

为了帮助读者理解上面所说的区别，我们用函数的组织形式来做个类比。不管是单体程序，还是多组件程序，系统架构都是由那些跨越架构边界的关键函数调用来定义的，并且整个架构必须遵守依赖关系规则。系统中许多其他的函数虽然也起到了隔离行为的效果，但它们显然并不具有架构意义。

服务的情况也一样，服务这种形式说到底不过是一种跨进程/平台边界的函数调用而已。有些服务会具有架构上的意义，有些则没有。我们这里重点要讨论的，当然是前者。

服务所带来的好处

我在本节的标题后面打了个问号，意味着我打算在这一节好好挑战一下目前流行的针对服务架构的崇拜情结。下面，就让我们针对那些所谓的好处，一个一个地来批驳。

解耦合的谬论

很多人认为将系统拆分成服务的一个最重要的好处就是让每个服务之间实现强解耦。毕竟，每个服务都是以一个不同的进程来运行的，甚至运行在不同的处理器上。因此，服务之间通常不能访问彼此的变量。其外，服务之间的接口一定是充分定义的。

从一定程度上来说，这是对的。确实，服务之间的确在变量层面做到了彼此隔离。然而，它们之间还是可能会因为处理器内的共享资源，或者通过网络共享资源而彼此耦合的。另外，任何形式的共享数据行为都会导致强耦合。

例如，如果给服务之间传递的数据记录中增加了一个新字段，那么每个需要操作这个字段的服务都必须要做出相应的变更，服务之间必须对这条数据的解读达成一致。因此其实这些服务全部是强耦合于这条数据结构的，因此它们是间接彼此耦合的。

再来说说服务能很好地定义接口——它确实能很好地定义接口——但函数也能做到这一点。事实上，服务的接口与普通的函数接口相比，并没有比后者更正式、更严谨，也没有更好，这一点根本算不上什么好处。

独立开发部署的谬论

人们认为的另一个使用服务的好处就是，不同的服务可以由不同的专门团队负责和运维。这让开发团队可以采用 dev-ops 混合的形式来编写、维护以及运维各自的服务，这种开发和部署上的独立性被认为是可扩展的。这种观点认为大型系统可以由几十个、几百个、甚至几千个独立开发部署的服务组成。整个系统的研发、维护以及运维工作就可以由同等量级的团队来共同完成。

这种理念有一些道理——但也仅仅是一些而已。首先，无数历史事实证明，大型系统一样可以采用单体模式，或者组件模式来构建，不一定非得服务化。因此服务化并不是构建大型系统的唯一选择。

其次，上文说到的解耦合谬论已经说明拆分服务并不意味着这些服务可以彼此独立开发、部署和运维。如果这些服务之间以数据形式或者行为形式相耦合，那么

它们的开发、部署和运维也必须彼此协调来进行。

运送猫咪的难题

下面，我们再以之前那个出租车调度系统为例来说明上面那两个谬论。各位还记得吗？该系统会负责统一调度给定城市中的多个出租车提供商，而用户可以集中在它那里下订单。在这里，我们假设用户在租车时往往会附带一组参考条件，例如接送时间、价格、豪华程度、司机的经验，等等。

我们希望整个系统是可扩展的，于是该系统大量采用了微服务架构。然后，我们进一步将整个研发团队划分为许多个小团队，每个团队都负责开发、维护和运维相应的小数量[1]的微服务。

这个虚构系统的架构如图 27.1 所示，整个系统都是依靠服务来构建的。`TaxiUI` 服务负责与用户打交道，用户会通过移动设备向它下订单。`TaxiFinder` 服务负责调用不同的 `TaxiSupplier` 服务来获取可用车辆的信息，并且找出可用的出租车以作为可推荐项。这些可推荐项会短期地被固化成一条数据记录，与用户信息挂钩。`TaxiSelector` 服务则负责根据用户所选择的价格、时间、豪华程度等条件从可选项中筛选结果，最后这些结果会被传递给 `TaxiDispatcher` 服务，由它负责分派订单。

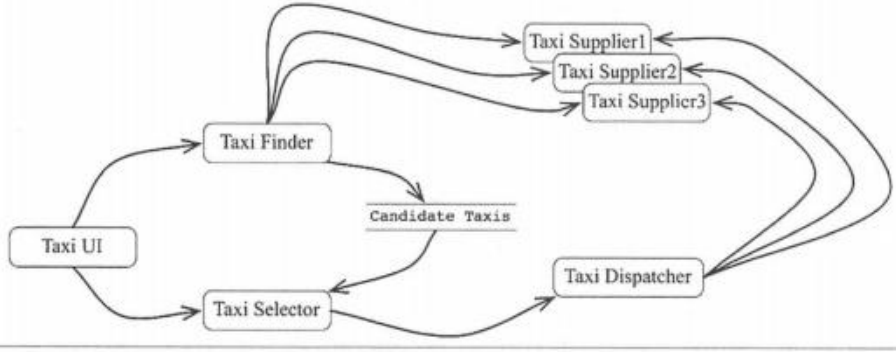


图 27.1：出租车调度系统的服务架构图

1 因此微服务的总数量应与程序员的数量大致相同。

现在，假设我们的系统已经运行了一年有余。其研发团队在持续开发新功能的同时，维护着所有的服务。

在一个阳光明媚的早上，市场部召集整个研发部开会。会议上，市场部宣布了他们在该城市开展猫咪送达服务的计划。该计划将允许用户向系统下订单，要求将他们的猫咪送到自己家里或者办公室。

公司的计划是在城市中建立几个猫咪集散点。当用户下订单时，附近的一辆出租车将被选中去集散点取猫，并将猫送到指定地点。

现在已经有一家出租车公司参加了这项活动，未来可能还会有其他公司参与进来，但肯定也会有不参与的公司。

当然，由于有些司机会对猫过敏，所以系统还必须要避免选中这些人去运送猫咪。同样的，由于出租车的乘客中也会有对猫过敏的人，所以当他们叫车时，系统也必须避免指派过去三天内运送过猫咪的车。

现在根据上述需求再来看我们的系统架构图，数一数有多少个服务需要变更？答案是全部！显然，为了增加这个运送猫咪的功能，该系统所有的服务都需要做变更，而且这些服务之间还要彼此做好协调。

换句话说，这些服务事实上全都是强耦合的，并不能真正做到独立开发、部署和维护。

这就是所谓的横跨型变更（cross-cutting concern）问题，它是所有的软件系统都要面对的问题，无论服务化还是非服务化的。其中，图 27.1 所示的这种按功能切分服务的架构方式，在跨系统的功能变更时是最脆弱的。

对象化是救星

如果采用组件化的系统架构，如何解决这个难题呢？通过对 SOLID 设计原则的仔细考虑，我们应该一开始就设计出一系列多态化的类，以应对将来新功能的扩展需要。

这种策略下的系统架构如图 27.2 所示，我们可以看到该图中的类与图 27.1 中的服务大致是相互对应的。然而，请读者注意这里设置了架构边界，并且遵守了依赖关系原则。

现在，原先服务化设计中的大部分逻辑都被包含在对象模型的基类中。然而，针对每次特定行程的逻辑被抽离到一个单独的 `Rides` 组件中。运送猫咪的新功能被放入到 `Kittens` 组件中。这两个组件覆盖了原始组件中的抽象基类，这种设计模式被称作模板方法模式或策略模式。

同时，我们也会注意到 `Rides` 和 `Kittens` 这两个新组件都遵守了依赖关系原则。另外，实现功能的类也都是由 UI 控制下的工厂类创建出来的。

显然，如果我们在这种架构下引入运送猫咪的功能，`TaxiUI` 组件就必须随之变更，但其他的组件就无须变更了。这里只需要引入一个新的 jar 文件或者 Gem、DLL。系统在运行时就会自动动态地加载它们。

这样一来，运送猫咪的功能就与系统的其他部分实现了解耦，可以实现独立开发和部署了。

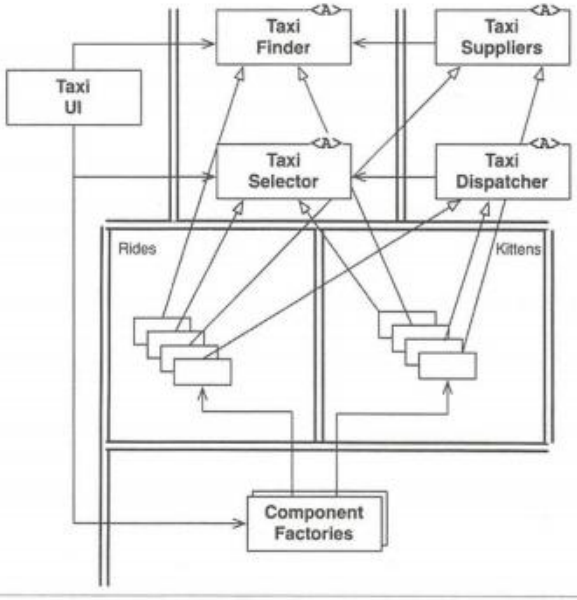


图 27.2：采用面向对象的方法来处理横跨型变更

基于组件的服务

那么，问题来了：服务化也可以做到这一点吗？答案是肯定的。服务并不一定必须是小型的单体程序。服务也可以按照 SOLID 原则来设计，按照组件结构来部署，这样就可以做到在添加/删除组件时不影响服务中的其他组件。

我们可以将 Java 中的服务看作是一个或多个 jar 文件中的一组抽象类，而每个新功能或功能扩展都是另一个 jar 文件中的类，它们都扩展了之前 jar 文件中的抽象类。这样一来，部署新功能就不再是部署服务了，而只是简单地在服务的加载路径下增加一个 jar 文件。换句话说，这种增加新功能的过程符合开闭原则（OCP）。

这种服务的架构如图 27.3 所示。我们可以看到，在该架构中服务仍然和之前一样，但是每个服务中都增加了内部组件结构，以便使用衍生类来添加新功能，而这些衍生类都有各自所生存的组件。

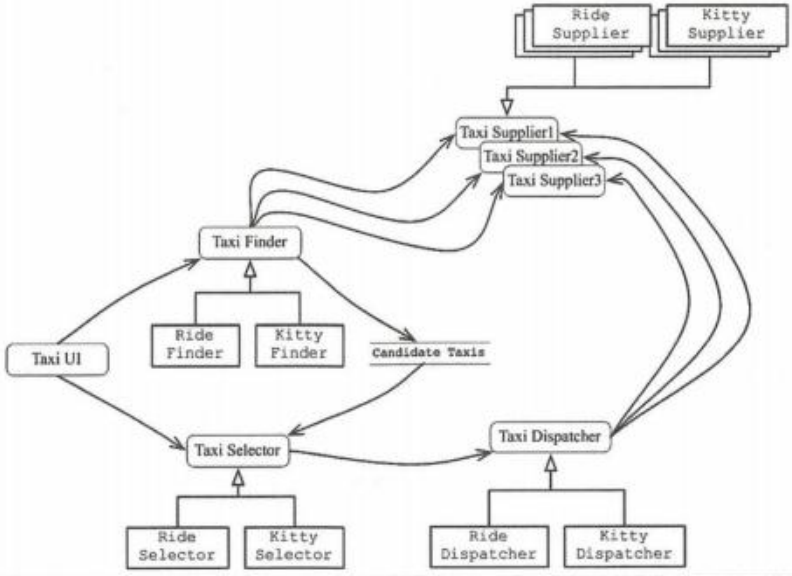


图 27.3：该架构中的每个服务有自己内部的组件结构，允许以衍生类的方式为其添加新功能

横跨型变更

现在我们应该已经明白了，系统的架构边界事实上并不落在服务之间，而是穿透所有服务，在服务内部以组件的形式存在。

为了处理这个所有大型系统都会遇到的横跨型变更问题，我们必须在服务内部采用遵守依赖关系原则的组件设计方式，如图 27.4 所示。总而言之，服务边界并不能代表系统的架构边界，服务内部的组件边界才是。

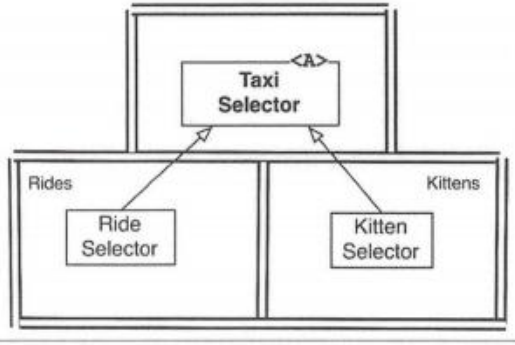


图 27.4：服务内部的组件的设计必须符合依赖指向规则

本章小结

虽然服务化可能有助于提升系统的可扩展性和可研发性，但服务本身却并不能代表整个系统的架构设计。系统的架构是由系统内部的架构边界，以及边界之间的依赖关系所定义的，与系统中各组件之间的调用和通信方式无关。

一个服务可能是一个独立组件，以系统架构边界的形式隔开。一个服务也可能由几个组件组成，其中的组件以架构边界的形式互相隔离。在极端情况下[1]，客户端和服务端甚至可能会由于耦合得过于紧密而不具备系统架构意义上的隔离性。

1 我们希望这是极端情况，但不幸的是事实上很常见。

第 28 章

测试边界

对，你没看错：和程序代码一样，测试代码也是系统的一部分。甚至，测试代码有时在系统架构中的地位还要比其他部分更独特一些。

测试也是一种系统组件

讨论测试的时候，业界总会有许多自相矛盾的声音。测试应该是系统的一部分吗？还是应该独立于系统之外存在呢？测试分为哪几种？单元测试与集成测试是不同的东西吗？质量检查测试、功能性测试、Cucumber 测试、TDD 测试、BDD 测试、组件测试分别又都是什么？

在本书中我们并不想卷入对这些问题的辩论中，而且也很庆幸没有卷入的必要。因为从架构的角度来讲，所有的测试都是一样的。不论它们是小型的 TDD 测试，还是大型的 FitNesse、Cucumber、SpecFlow 或 JBehave 测试，对架构来说都是一样的。

究其本质而言，测试组件也是要遵守依赖关系原则的。因为其中总是充满了各种细节信息，非常具体，所以它始终都是向内依赖于被测试部分的代码的。事实上，我们可以将测试组件视为系统架构中最外圈的程序。它们始终是向内依赖的，而且系统中没有其他组件依赖于它们。

另外，测试组件是可以独立部署的。事实上，大部分测试组件都是被部署在测试环境中，而不是生产环境中的。所以，即使是在那些本身不需要独立部署的系统中，其测试代码也总是独立部署的。

测试组件通常是一个系统中最独立的组件。系统的正常运行并不需要用到测试组件，用户也不依赖于测试组件。测试组件的存在是为了支持开发过程，而不是运行过程。然而，测试组件仍然是系统中不可或缺的一个组件。事实上，测试组件在许多方面都反映了系统中其他组件所应遵循的设计模型。

可测试性设计

由于测试代码的独立性，以及往往不会被部署到生产环境的特点，开发者常常会在系统设计中忽视测试的重要性，这种做法是极为错误的。测试如果没有被集成到系统设计中，往往是非常脆弱的，这种脆弱性会使得系统变得死板，非常难以更改。

当然，这里的关键之处就是耦合。如果测试代码与系统是强耦合的，它就得随着系统变更而变更。哪怕只是系统中组件的一点小变化，都可能会导致许多与之相耦合的测试出现问题，需要做出相应的变更。

这个问题可能会很严重。修改一个通用的系统组件可能会导致成百上千个测试出现问题，我们通常将这类问题称为*脆弱的测试问题*（*fragile tests problem*）。

这类问题的发生过程并不难想象。例如，假设我们有一套利用 GUI 来校验系统业务逻辑的测试。这些测试可能从登录页面开始，按照导航顺序遍历整个页面结构，直到完成某个特定的业务逻辑为止。这时候，任何针对登录页面或导航顺序的变更，都可能导致大量的测试出错。

另外，脆弱的测试还往往会让系统变得非常死板。当开发者意识到一些简单的修改就会导致大量的测试出错时，他们自然就会抵制修改。请想象一下，如果市场部门所要求的一个针对页面导航结构的简单变更会导致一千个测试出错时，开发部门会怎么说吧。

要想解决这个问题，就必须在设计中考虑到系统的可测试性。软件设计的第一条原则——不管是为了可测试性还是其他什么东西——是不变的，就是*不要依赖于多变的东西*。譬如，GUI 往往是多变的，因此通过 GUI 来验证系统的测试一定是*脆弱的*。因此，我们在系统设计与测试设计时，应该让业务逻辑不通过 GUI 也可以被测试。

测试专用 API

设计这样一个系统的方法之一就是专门为验证业务逻辑的测试创建一个 API。这个 API 应该被授予超级用户权限，允许测试代码可以忽视安全限制，绕过那些成本高昂的资源（例如数据库），强制将系统设置到某种可测试的状态中。总而言之，该 API 应该成为用户界面所用到的*交互器*与*接口适配器*的一个超集。

设置测试 API 是为了将测试部分从应用程序中分离出来。换句话说，这种解耦动作不只是为了分隔测试部分与 UI 部分，而是要将测试代码的*结构*与应用程序其他部分的*代码结构*分开。

结构性耦合

结构性耦合是测试代码所具有的耦合关系中最强大、最阴险的一种形式。假设我们现在有一组测试套件，它针对每个产品类都有一个对应的测试类，每个产品函数都有一个对应的测试函数。显然，该测试套件与应用程序在结构上是紧耦合的。

每当应用程序中的一个函数或类发生变更时，该测试套件就必须进行大量相应的修改。因此，这些测试是非常脆弱的，它们也会让产品代码变得非常死板。

测试专用 API 的作用就是将应用程序与测试代码解耦。这样，我们的产品代码就可以在不影响测试的情况下进行重构和演进。同样的，这种设计也允许测试代码在不影响生产代码的情况下进行重构和演进。

这种对演进过程的隔离是很重要的，因为随着时间的推移，测试代码趋向于越来越具体和详细。相比之下，我们的产品代码则会趋向于越来越抽象和通用。结构性的强耦合可能会让这种必需的演进无法进行——至少会形成强烈的干扰。

安全性

当然，这种具有超级权限的测试专用 API 如果被部署到我们的产品系统中，可能会是非常危险的。如果要避免这种情况发生，应该将测试专用 API 及其对应的具

体实现放置在一个单独的、可独立部署的组件中。

本章小结

测试并不是独立于整个系统之外的，恰恰相反，它们是系统的一个重要组成部分。我们需要精心设计这些测试，才能让它们发挥验证系统稳定性和预防问题复发的作用。没有按系统组成部分来设计的测试代码，往往是非常脆弱且难以维护的。这种测试最后常常会被抛弃，因为它们终究会出问题。

第 29 章

整洁的嵌入式架构

前一段时间，我在 Doug Schmidt 的个人博客上看到了一篇文章，标题是“The Growing Importance of Sustaining Software for the DoD”[1]，Doug 在这篇文章中提出了以下观点：

> “虽然软件本身并不会随时间推移而磨损，但硬件及其固件却会随时间推移而过时，随即也需要对软件做相应改动。”

这句话对我有如醍醐灌顶。Doug 在这里用到了两个专业名词，我一直认为是显而易见的，但是其他人可能并没有这么觉得。其中，软件（*software*）应该是一种使用周期很长的东西，而固件（*firmware*）则会随着硬件演进而淘汰过时。曾经开发过嵌入式系统的人一定都知道，硬件系统是在持续不断地演进的。与此同时，随着新功能不断地增加，软件复杂度也在不断上升。

在这里，我想对 Dough 上面的那个观点做一点补充：

> “虽然软件质量本身并不会随时间推移而损耗，但是未妥善管理的硬件依赖和固件依赖却是软件的头号杀手。”

也就是说，本可以长期使用的嵌入式软件可能会由于其中隐含的硬件依赖关系而无法继续使用，这种情况是很常见的。

我个人很喜欢 Doug 对固件所做的定义，但我们也可以来看一下其他人对固件的定义，以下是我目前所找到的一些说法：

- “固件通常被存储在非可变内存设备，例如 ROM、EPROM 或者闪存中。”（https://en.wikipedia.org/wiki/Firmware）
- “固件是直接编程在一个硬件设备上的一组指令或者一段程序。”（https://techterms.com/definition/firmware）
- “固件是嵌入在一个硬件中的软件程序。”（https://www.lifewire.com/what-is-firmware-2625881）

1 参见 https://insights.sei.cmu.edu/sei_blog/2011/08/the-growing-importance-of-sustaining-software-for-thedod.html。

- “固件是被写入到只读内存设备中的（ROM）程序或数据。”
  （http://www.webopedia.com/TERM/F/firmware.html）

Doug 的这段观点表述让我意识到，大家普遍所认知的固件定义是错误的，或者至少是过时的。固件并不一定是指存储在 ROM 中的代码。固件也并不是依据其存储的位置来定义的，而是由其代码的依赖关系，及其随着硬件的演进在变更难度上的变化来定义的。硬件的演进是显而易见的（如果对此有任何疑问，请想一想你手中的手机），我们在架构嵌入式代码时要时刻记住这一点。

我并不反对固件，也不反对固件工程师（我自己也曾经写过固件）。但是我们真的应该少写点固件，而多写点软件。事实上，我最失望的是固件工程师竟然要写那么多固件程序！

还有，非嵌入式工程师竟然也要写固件程序！虽然你可能并不是嵌入式系统的开发者，但如果你在代码中嵌入了 SQL 或者是代码中引入了对某个平台的依赖的话，其实就是在写固件代码。譬如，Android 工程师在没有将业务逻辑与 Android API 分离之前，实际上也是在写固件代码。

我参与过很多软件项目，其中一些产品的功能代码（软件）与硬件支持代码（固件）的边界模糊得几乎不存在。例如，在 20 世纪 90 年代末，我有幸参与了一套通信系统的重新设计，将其从时分复用（TDM）模式迁移到 VOIP 模式。虽然 VOIP 如今已经是行业标准，但是从 20 世纪 50 年代到 60 年代，TDM 一直都是非常先进的技术，直到 20 世纪 80 年代和 90 年代它也被广泛部署在各种系统中。

每当我们向系统工程师提出一个产品问题——系统在某个情况下应该如何处理某通电话，这位系统工程师就会消失一段时间，然后给出一个非常具体的答案。我们问他是从哪里查到这个结果的？答案是“从当前的产品代码里！”这些复杂交错的老旧代码已经成为系统定义的一部分。该系统在实现过程中并没有区分 TDM 技术代码和拨打电话这样的业务逻辑代码。整个产品从头到尾都与具体技术、具体硬件息息相关，无法分割。可以说整个产品已经成为事实上的固件。

再来看另外一个例子：我们都知道命令消息是通过串行端口传递给系统的。这自然就要有一个消息的处理器/分发器系统。其中，消息处理器得了解消息格式，可以解析消息，然后将消息分发给具体的处理代码。这些都很正常，但消息处理器/分

发器的代码和操作 UART 硬件[1]的代码往往会被放在同一个文件中，消息处理器的代码中常常充斥着与 UART 相关的实现细节。这样一来，本可以长时间使用的消息处理器代码变成了一段固件代码，这太不应该了！

虽然我意识到要从使用意义上将软件与硬件、固件区分开来也已经有一段时间了，但是借助 Doug 对软件和固件的定义，我现在可以把这件事情说得更明白一些。

对于程序员和工程师，我的意思很明确：不要再写固件代码了，让我们的代码活得更久一点！当然，我们也不能总是空谈理念，下面就来看一下应该如何通过好的架构设计让嵌入式代码拥有更长的有效生命周期。

“程序适用测试”测试

为什么这么多嵌入式软件最后都成了固件？看起来，很可能是因为我们在做嵌入式设计时只关注代码能否顺利运行，并不太关心其结构能否撑起一个较长的有效生命周期。Kent Beck 描述了软件构建过程中的三个阶段（引号部分是他的原话，楷体部分是我的注解）：

1. “先让代码工作起来”——*如果代码不能工作，就不能产生价值。*

2. “然后再试图将它变好”——*通过对代码进行重构，让我们自己和其他人更好地理解代码，并能按照需求不断地修改代码。*

3. “最后再试着让它运行得更快”——*按照性能提升的“需求”来重构代码。*

我所见过的大部分“野生”的嵌入式代码，都只关注“先让它工作起来”这个目标——也许还有些团队会同时痴迷于“让它更快”这个目标，不放过任何一个机会加入各种微优化。在《人月神话》这本书中，Fred Brooks 建议我们应该随时准备“抛弃一个设计”。Kent 和 Fred 说的其实是同一件事：“在实践中学习正确的工作方法，然后再重写一个更好的版本”。

这个建议对非嵌入式软件系统开发同样有用。毕竟目前大部分非嵌入式应用也

1　这是一种用来控制串行端口的硬件设备。

仅仅停留在“可用”这个目标上，很少考虑为了长久使用而进行正确的设计。

对于程序员来说，让他的程序工作这件事只能被称为“程序适用测试（*app-titude test*）”。一个程序员，不论他写的是否是嵌入式程序，如果目标仅仅是让程序可以工作，恐怕对他的老板和这个程序本身而言都是一件坏事。毕竟，编程这件事可远不止是让程序可以工作这么简单。

下面我们来示范一下可以通过“程序适用测试”的代码是什么样子的。先来看一个小型嵌入式系统中某个源文件中的一段函数声明：

```
ISR(TIMER1_vect) { ... }
ISR(INT2_vect) { ... }
void btn_Handler(void) { ... }
float calc_RPM(void) { ... }
static char Read_RawData(void) { ... }
void Do_Average(void) { ... }
void Get_Next_Measurement(void) { ... }
void Zero_Sensor_1(void) { ... }
void Zero_Sensor_2(void) { ... }
void Dev_Control(char Activation) { ... }
char Load_FLASH_Setup(void) { ... }
void Save_FLASH_Setup(void) { ... }
void Store_DataSet(void) { ... }
float bytes2float(char bytes[4]) { ... }
void Recall_DataSet(void) { ... }
void Sensor_init(void) { ... }
void uC_Sleep(void) { ... }
```

可以看到该源文件中的函数是按一定顺序列出来的。现在我们要按照功能进行分组：

- 用于定义域逻辑（domain logic）的函数

```
float calc_RPM(void) { ... }
void Do_Average(void) { ... }
void Get_Next_Measurement(void) { ... }
void Zero_Sensor_1(void) { ... }
void Zero_Sensor_2(void) { ... }
```


- 用于设置硬件平台的函数

```
ISR(TIMER1_vect) { ... }*
ISR(INT2_vect) { ... }
void uC_Sleep(void) { ... }
响应开关按钮动作的函数
void btn_Handler(void) { ... }
void Dev_Control(char Activation) { ... }
用于从硬件中获取 A/D 输入读数的函数
static char Read_RawData(void) { ... }
```

- 用于执行持久化存储的函数

```
char Load_FLASH_Setup(void) { ... }
void Save_FLASH_Setup(void) { ... }
void Store_DataSet(void) { ... }
float bytes2float(char bytes[4]) { ... }
void Recall_DataSet(void) { ... }
```

- 功能与其名字不符的函数

```
void Sensor_init(void) { ... }
```

在查看该应用程序其他源代码文件的过程中，我同样发现了许多理解上的障碍点。同时，我还发现这个项目的结构决定了该应用程序的所有代码只有在指定硬件平台上才能被测试。几乎代码的所有部分都知道它要运行在一个特殊的微处理器平台上，因为它们用的是“被扩展了的”C 结构[1]，需要特殊的工具链和微处理器才能执行。除非这个产品永远不需要迁移到另一个硬件平台上，否则这段代码几乎不可能有长久的使用价值。

所以你看，这段代码的确能够正常工作：它的工程师也通过了“程序适用性测试”，但我们不能说该应用程序有一套整洁的嵌入式架构。

1　有些集成芯片会给 C 语言增加一些特殊的关键字，以简化访问寄存器和 I/O 端口的 C 代码。然而，这样做的话，代码就不再符合 C 语言的标准了。

目标硬件瓶颈

嵌入式系统的程序员通常需要处理很多在写非嵌入式系统时不需要关心的事情——例如，有限的地址空间、实时性限制、运行截止时间、有限的 I/O 能力、非常规的用户接口、感应器，以及其他与物理世界的实际链接。大部分时候，这些系统的硬件是和它的软件、固件并行开发的。工程师在为这种系统编写代码的时候，往往没有任何地方可以运行。如果你认为这还不算糟糕的话，请想象一下，如果我们要等到真正拿到硬件时，才能了解代码在该硬件上存在着哪些意料之外的缺陷，这会在多大程度上拖慢我们的开发进度？

是的，我们承认嵌入式系统的开发有其特殊性，嵌入式工程师的工作有其特殊性，但我们并不认为嵌入式开发特殊到本书所讲的原则都不适用的程度。

目标硬件瓶颈（*target-hardware bottleneck*）是嵌入式开发所特有的一个问题，如果我们没有采用某种清晰的架构来设计嵌入式系统的代码结构，就经常会面临只能在目标系统平台上测试代码的难题。如果只能在特定的平台上测试代码，那么这一定会拖慢项目的开发进度。

整洁的嵌入式架构就是可测试的嵌入式架构

下面，我们来看一下具体应如何将架构设计的原则应用在嵌入式软件和固件上，以避免陷入目标硬件瓶颈。

分层

分层可以有很多种方式，这里先按图 29.1 所示的设计将系统分成三层。首先，底层是硬件层。正如 Doug 警告我们的那样，由于科技的进步与摩尔定律，硬件是一定会改变的。旧的硬件部件将会被淘汰，新的硬件部件可能耗电量更少，或者性能更好，或者价格更便宜。不管硬件更新的原因是什么，作为嵌入式工程师，我们都不会希望这些不可避免的硬件变动带来更多的工作量。

图 29.1：三层结构设计

硬件与系统其他部分的分隔是既定的——至少在硬件设计完成之后如此（如图 29.2 所示）。这也是在我们试图通过程序适用测试之时往往会发生问题的地方。因为没有什么东西可以真正阻碍硬件实现细节污染到应用代码。如果我们在构建代码的时候不够小心，没有小心安排哪些模块之间可以互相依赖，代码很快就非常难以更改了。请注意，这里所说的变更不仅仅是指来自硬件的变更，还包括用户的功能性变更、修复代码中的 Bug。

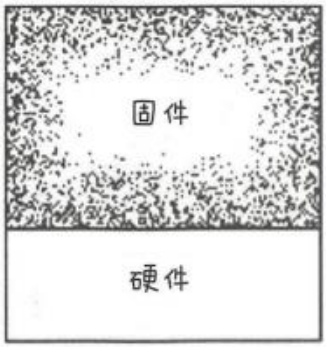


图 29.2：硬件必须与系统其他部分分隔开

另外，软件与固件集成在一起也属于设计上的反模式（anti-pattern）。符合这种反模式的代码修改起来都会很困难。同时，这种代码也很危险，容易造成意外事故，这导致它经历任何微小的改动都需要进行完整的回归测试。如果没有完善的测试流程，那么你就等着无穷无尽的手工测试吧——同时还有纷沓而来的 Bug 报告。

硬件是实现细节

软件与固件之间的分割线往往没有代码与硬件之间的分割线那么清晰，如图 29.3 所示。

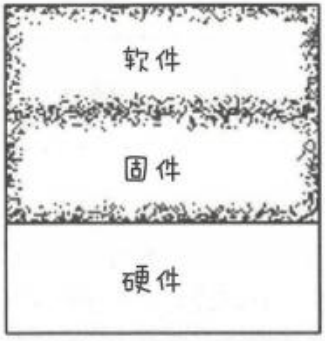


图 29.3：软件与固件之间的边界往往没有代码与硬件之间的边界那么清晰

所以，我们的工作之一就是将这个边界定义得更清晰一些。软件与固件之间的边界被称为硬件抽象层（HAL），如图 29.4 所示。这不是一个新概念，它在 PC 上的存在甚至可以追溯到 Windows 诞生之前。

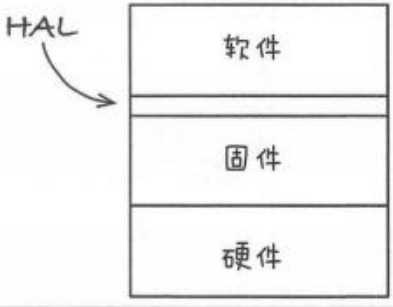


图 29.4：硬件抽象层

HAL 的存在是为了给它上层的软件提供服务，HAL 的 API 应该按照这些软件的需要来量身定做。例如，固件可以直接将字节和字节组存入闪存中。相比之下，软件需要的是从某种持久化平台保存和读取 name/value 对信息，它不应该关心自己信息到底是被存储到闪存中、磁盘中、云端存储中，还是在内存中读取/存储这些信息。总之，HAL 的作用是为软件部分提供一种服务，以便隐藏具体的实现细节。毕

竟专门针对闪存的实现代码是一种细节信息，它应该与软件部分隔离。

我们再来看另一个例子：有一个 LED 被连接到一个 GPIO 比特位上。固件可以直接操作 GPIO 比特位，而 HAL 则会提供一个 `Led_TurnOn(5)` 数，这种硬件抽象层的层次是相当低的。现在，假设我们想将抽象层次从硬件层次提升到软件/产品的层次。这时候就要弄清楚这个 LED 到底代表的是什么。假设它代表了电池电量不足，那么其固码（或该电路板的支持包）可能就会负责提供 `Led_TurnOn(5)` 函数，而 HAL 则负责提供 `Indicate_LowBattery()` 函数。由此可见，HAL 层是按照应用程序的需要来提供服务的。同时，我们也能看出来系统的每一个分层中都可以包含许多分层。相对于之前的固定分层法，这里更像是一种无限分层模式。总之，GPIO 位的对应关系应该是一个具体的实现细节，它应该与软件部分隔离。

不要向 HAL 的用户暴露硬件细节

依照整洁的嵌入式架构所建构的软件应该是可以脱离目标硬件平台来进行测试的。因为设计合理的 HAL 可以为我们脱离硬件平台的测试提供相应的支撑。

处理器是实现细节

当我们的嵌入式应用依赖于某种特殊的工具链时，该工具链通常会为我们提供一些“帮助”[1]性质的头文件。这些编译器往往会自带一些基于 C 语言的扩展库，并添加一些用于访问特殊功能的关键词。这会导致这些程序的代码看起来仍然用的是 C 语言，但实际上它们已经不是 C 语言了。

有时候，这些嵌入式应用的提供商所指定的 C 编译器还会提供类似于全局变量的功能，以便我们直接访问寄存器、I/O 端口、时钟信息、I/O 位、中断控制器以及其他处理器函数，这些函数会极大地方便我们对相关硬件的访问。但请注意，一旦你在代码中使用了这些函数，你写的就不再是 C 语言程序，它就不能用其他编译器来编译了，甚至可能连同一个处理器的不同编译器也不行。

我不想说这是提供商故意给我们设置的陷阱，即使我们假设这些硬件提供商这

1　这里使用 HTML 代码是故意的。

样做真的是为了“帮助”我们，我们自己也要知道如何来利用这些“帮助”，这才是问题的关键。为了避免自己的代码在未来出现问题，我们就必须限制这些 C 扩展的使用范围。

下面来看一下针对 ACME DSP（数字信号处理器）系统设计的头文件——Wile E Coyote 采用的就是这个系统：

```
#ifndef _ACME_STD_TYPES
#define _ACME_STD_TYPES

#if defined(_ACME_X42)
    typedef unsigned int            Uint_32;
    typedef unsigned short          Uint_16;
    typedef unsigned char           Uint_8;

    typedef int                     Int_32;
    typedef short                   Int_16;
    typedef char                    Int_8;

#elif defined(_ACME_A42)
    typedef unsigned long           Uint_32;
    typedef unsigned int            Uint_16;
    typedef unsigned char           Uint_8;

    typedef long                    Int_32;
    typedef int                     Int_16;
    typedef char                    Int_8;
#else
    #error <acmetypes.h> is not supported for this environment
#endif

#endif
```

该 acmetypes.h 头文件通常不应该直接使用。因为如果这样做的话，代码就和某个 ACME DSP 绑定在一起了。这时候你可能会问，我们在这里写代码不就是为了使用 ACME DSP 吗？不引用这个头文件如何编译代码呢？但如果引用了这个头文件，就等于同时定义了__ACME_X42 和__ACME_A42，那么我们的代码在平台之

外进行测试的时候整数类型的大小就会是错误的。更糟糕的是，有一天当我们想将代码迁移到另外一个处理器上的时候，如果没有在这里限制 ACME 头文件被引用的范围，就会大大增加这项迁移工作的难度。

因此在这里，我们应该用更标准的 stdint.h 来替代 acmetypes.h。如果目标编译器没有提供 stdint.h 的话，我们可以自己写一个。例如，下面就是一个针对目标编译器的，可以用 acmetypes.h 来构建目标的自定义 stdint.h：

```
#ifndef _STDINT_H_
#define _STDINT_H_

#include <acmetypes.h>

typedef Uint_32 uint32_t;
typedef Uint_16 uint16_t;
typedef Uint_8 uint8_t;

typedef Int_32 int32_t;
typedef Int_16 int16_t;
typedef Int_8 int8_t;

#endif
```

使用 stdint.h 来编写嵌入式的软件和固件，你的代码会是整洁且可移植的。当然，我们应该让所有的软件都独立于处理器，但这并不是所有固件都可以做到的。下面这段代码使用了特殊的 C 扩展来访问微处理器的配件，这样做的目的很可能就是为了使用这个配件。这个函数的作用就是输出一行"hi"到串口。（该例子来自于一个真实项目。）

```
void say_hi()
{
  IE = 0b11000000;
  SBUF0 = (0x68);
  while(TI_0 == 0);
  TI_0 = 0;
  SBUF0 = (0x69);
  while(TI_0 == 0);
```


```
    TI_0 = 0;
    SBUF0 = (0x0a);
    while(TI_0 == 0);
    TI_0 = 0;
    SBUF0 = (0x0d);
    while(TI_0 == 0);
    TI_0 = 0;
    IE = 0b11010000;
}
```

这个小函数中存在大量的问题。首先，你注意到的可能就是 0b11000000。二进制表示法的确很酷，但 C 语言支持它吗？并不支持。另外还有一些问题也与 C 语言的扩展有关。

IE：设置中断比特位。

SBUF0：串口输出缓冲区。

TI_0：串口传输区空中断。读取到 1 表明缓冲区为空。

这些名字大写的变量实际上都是用来访问微处理器的内置部件的。如果我们需要控制中断并且输出字符，就必须使用这些部件，它们也确实很方便——但这不是 C 代码。

在整洁的嵌入式架构中，我们会将这些用于设备访问的寄存器访问集中在一起，并将其限制在固件层中。这样一来，任何需要知道这些寄存器值的代码都必须成为固件代码，与硬件实现绑定。一旦这些代码与处理器实现强绑定，那么在处理器稳定工作之前它们是无法工作的，并且在需要将其迁移到一个新处理器上时也会遇到麻烦。

如果我们真的需要使用这种微处理器，固件就必须将这类底层函数隔离成处理器抽象层（PAL），这样一来，使用 PAL 的固件代码就可以在目标平台之外被测试了。

操作系统是实现细节

HAL 的必要性是不言而喻的，但光有 HAL 就够了吗？在裸机的嵌入式系统中，HAL 的确可能可以保证我们的代码不会和运行环境绑定得太紧密。但如果嵌入式系统使用了某种实时操作系统（RTOS），或者某种嵌入式的 Linux 或 Windows 呢？

为了延长代码的生命周期，我们必须将操作系统也定义为实现细节，让代码避免与操作系统层产生依赖。

软件通过操作系统来访问运行环境服务。操作系统是将软件与固件隔离的那一层（见图 29.5），直接使用操作系统服务可能会带来问题。例如，如果更换了 RTOS 操作系统厂商，授权费用提高，或者质量下降怎么办？如果需求发生变化，RTOS 无法满足怎么办？很多代码都需要变动，不仅要更改语法适应新操作系统 API，很有可能需要重新适应新操作系统的语义与原语。

图 29.5：添加操作系统层

整洁的嵌入式架构会引入操作系统抽象层（OSAL，如图 29.6 所示），将软件与操作系统分隔开。在某些情况下，实现这个抽象层就像给函数改个名字那么简单。而在另一些情况下，则需要将几个函数封装在一起。

如果你有过迁移 RTOS 系统的经历，就一定知道那有多痛苦。如果我们能让自己的软件依赖于 OSAL，而不是直接依赖于操作系统，我们就只需要写一个兼容以

前的 OSAL 实现的新版本即可。你觉得哪一种方式更好？是修改一堆复杂的现有代码，还是按照接口和行为定义来写一套新代码？这里显而易见，后者更好。

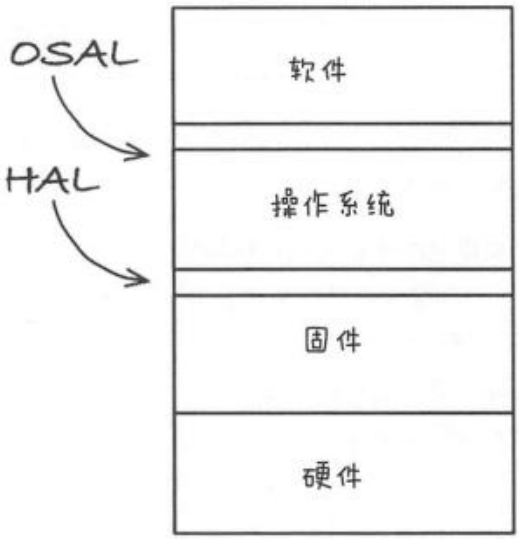


图 29.6：操作系统抽象层

当然，我们可能会担心代码膨胀的问题。但是，其实上面这种分层已经将因为使用操作系统所带来的重复性代码隔离开了，因此这种重复不一定会带来很大的额外负担。而且，如果我们定义了 OSAL，还可以让自己的应用共享一种公用结构。比如采用一套标准的消息传递机制，这样每个线程就不用自己定义一个并行模型了。

另外，OSAL 还可以帮助高价值的应用程序实现在目标平台、目标操作系统之外进行测试。一个由整洁的嵌入式架构所构建出来的软件是可以在目标操作系统之外被测试的。设计良好的 OSAL 会为这种目标环境外的测试提供支撑点。

面向接口编程与可替代性

除了在嵌入式系统的主要分层（指软件、操作系统、固件、硬件这四层）之中增加 HAL 和 OSAL 之外，我们还可以——也应该——应用本书中提到的其他设计原则。这些设计原则可以帮助我们按功能模块、接口编程以及可替代性来划分系统。

分层架构的理念是基于接口编程的理念来设计的。当模块之间能以接口形式交

互时，我们就可以将一个服务替换成另外一个服务。例如，很多读者应该都写过能在某个目标机器上运行的、小型的自定义的 printf 函数。只要我们的 printf 与标准的 printf 函数接口一致，它们就可以互相替换。

目前的普适规则之一就是用头文件来充当接口的定义。然而，如果真的要这样做的话，就需要小心控制头文件中的内容，尽量确保头文件中只包括函数声明，以及函数所需要的结构体名字和常量。

另外，不要在定义接口的头文件中包含只有具体实现代码才需要的数据结构、常量以及类型定义（typedef）。这不仅仅是架构是否整洁的问题，而是这样做可能会导致意外的依赖关系。总之，我们必须控制好实现细节的可见性，因为这些实现细节是肯定会变化的。关注实现细节的代码越少，它们所需的变更就越少。

由整洁的嵌入式架构所构建的系统应该在每一个分层中都是可测试的，因为它的模块之间采用接口通信，每一个接口都为平台之外的测试提供了替换点。

DRY 条件性编译命令

另一个经常被忽视的可替代换性规则的实际案例是嵌入式 C/C++程序对不同平台和操作系统的处理方式。这些程序经常会用条件性编译命令来根据不同的平台启用和禁用某一段代码。例如，我曾经遇到过 #ifdef BOARD_V2 这条语句在一个电信应用程序中出现了几千次的情况。

很显然，这种代码的重复违背了“不要重复自己（DRY）”原则[1]。如果#ifdef BOARD_V2 只出现一次，这当然不是什么问题，而如果出现了 6000 次，那就非常严重了。但这类条件性编译语句在嵌入式编程中非常常见，有什么好的解决方案吗？

使用硬件抽象层如何？这样的话，硬件类型就只是 HAL 中的一个实现细节了。而且，如果系统中使用的是 HAL 所提供的一系列接口，而不是条件性编译语句，那么我们就可以用链接器，或者某种运行时加载器来将软件与硬件相结合了。

1　请参考 Hunt 和 Thomas 合著的 *The Pragmatic Programmer* 一书。

本章小结

嵌入式编程人员应该多学习一些非嵌入式系统的编程经验。如果你从事的是嵌入式编程工作，相信你一定会从本章的建议中得到很多启发。

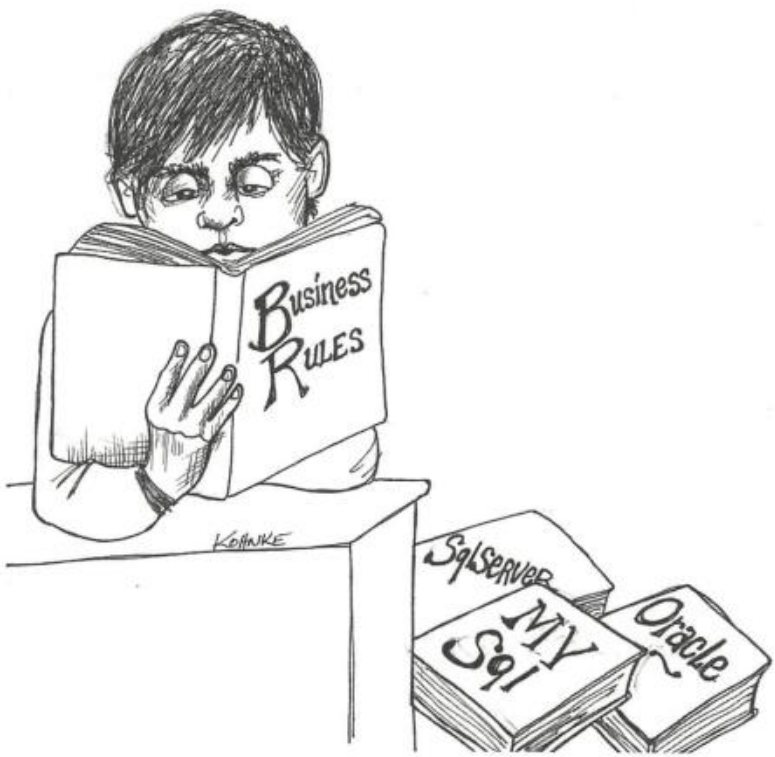
为了让我们的产品能长期地保持健康，请别让你的代码都变成固件。如果一个系统的代码只能在目标硬件上测试，那么它的开发过程会变得非常艰难。总之，为产品的长期健康着想而采用一套整洁的嵌入式架构是很有必要的。

第 6 部分

实现细节

第 30 章

数据库只是实现细节



从系统架构的角度来看，数据库并不重要——它只是一个实现细节，在系统架构中并不占据重要角色。如果就数据库与整个系统架构的关系打个比方，它们之间就好比是门把手和整个房屋架构的关系。

这个比喻肯定会招来非议。相信我，这种架我吵过很多次了。所以我在这里要把话说得清楚一点：这里讨论的不是数据模型。为应用程序中的数据设计结构，对于系统架构来说当然是很重要的，但是数据库并不是数据模型。数据库只是一款软件，是用来存取数据的工具。从系统架构的角度来看，工具通常是无关紧要的——因为这只是一个底层的实现细节，一种达成目标的手段。一个优秀的架构师是不会让实现细节污染整个系统架构的。

关系型数据库

关系型数据库的基本原理是 Edgar Codd 在 1970 年定义的。到了 20 世纪 80 年代中期，这种关系模型已经成为数据存储设计的主流。由于关系模型优雅、自律、非常稳健，因此得到了非常广泛的应用。总之，关系型数据库是一种非常优秀的数据存储与访问技术。

但不管关系型数据库的设计有多么有智慧，多么精巧，多么符合数学原理，它仍然也只是一种技术。换句话说，它终究只是一种实现细节。

虽然关系型数据的表模型设计对某一类数据访问需要来说可能很方便，但是把数据按行组织成表结构本身并没有什么系统架构意义上的重要性。应用程序的用例不应该知道，也不应该关心这么低层次的实现细节，需要了解数据表结构的代码应该被局限在系统架构的最外圈、最低层的工具函数中。

很多数据访问框架允许将数据行和数据表以对象的形式在系统内部传递。这么做在系统架构上来说是完全错误的，这会导致程序的用例、业务逻辑、甚至 UI 与数据的关系模型相互绑定在一起。

为什么数据库系统如此流行

为什么数据库系统在软件系统和企业软件领域如此流行？Oracle、MySQL 和 SQL Server 这些产品广泛流行的原因是什么？答案是硬盘。

带有高速旋转的盘片，以磁感应方式读取数据的硬盘在过去五十年成为数据存储的主流手段，以至于最近几代软件工程师对其他类型的数据存储几乎一无所知。而且硬盘技术一直在发展，早先一大摞重达数吨的直径 48 英寸的盘片只能存储 20 兆字节，现在单个直径 3 英寸、重量仅仅几克的薄薄的一张硬盘就能存储上 TB 的数据。这发展得实在是太快了！但是在硬盘的整个发展过程中，程序员们始终被一个限制困扰着：磁盘的访问速度太慢了！

在磁盘上，数据是按照环形轨道存储的。这些轨道又会进一步被划分成一系列扇区，这些扇区的大小通常是 4 KB。而每个盘片上都有几百条轨道，整个硬盘可能由十几个盘片组成。如果要从硬盘上读取某一个特定字节，需要将磁头挪到正确的轨道上，等待盘片旋转到正确的位置上，再将整个扇区读入内存中，从内存中查询对应的字节。这些过程当然需要时间，所以硬盘的访问速度一般在毫秒级。

毫秒级的速度看起来好像并不是很慢，但这已经比大多数处理器的速度慢一百万倍了。如果数据不在硬盘上，访问速度通常就通常是纳秒级，而不是毫秒级了。

为了应对硬盘访问速度带来的限制，必须使用索引、缓存以及查询优化器等技术。同时，我们还需要一种数据的标准展现格式，以便让索引、缓存及查询优化器来使用。概括来说，我们需要的就是某种数据访问与管理系统。过去几十年内，业界逐渐发展出了两种截然不同的系统：文件系统与关系型数据库系统（RDBMS）。

文件系统是基于文档格式的，它提供的是一种便于存储整个文档的方式。当需要按照名字存储数据和查找一系列文档时，文件系统很有用，但当我们需要检索文档内容时，它就没那么有用了。也就是说，我们在文件系统中查找一个名字为 `login.c` 的文件很容易，但要检索出所有包括变量 `x` 的 `.c` 文件就很困难，速度也很慢。

而数据库系统则主要关注的是内容，它提供的是一种便于进行内容检索的存储方式。其最擅长的是根据某些共同属性而检索一系列记录。然而，它对存储和访问内容不透明的文档的支持就没那么强了。

这两种系统都是为了优化磁盘存储而设计的，人们需要根据它们的特点来将数据组织成最便于访问的模式。每个系统都有一套索引和安排数据的方式。同时，每种系统最终都会将数据缓存在内存中，方便快速操作。

假设磁盘不存在会怎样

虽然硬盘现在还是很常见，但其实已经在走下坡路了。很快它们就会和磁带、软盘、CD 一样成为历史，RAM 正在替代一切。

现在，我们要来考虑一下：如果所有的数据都存在内存中，应该如何组织它们呢？需要按表格存储并且用 SQL 查询吗？需要用文件形式存储，然后按目录查找吗？

当然不，我们会将数据存储为链表、树、哈希表、堆栈、队列等各种各样的数据结构，然后用指针或者引用来访问这些数据——*因为这对程序员来说是最自然的方式。*

事实上，如果你再仔细想想，就会发现我们已经在这样做了。即使数据保存在数据库或者文件系统中，我们最终也会将其读取到内存中，并按照最方便的形式将其组织成列表、集合、堆栈、队列、树等各种数据结构，继续按文件和表格的形式来操作数据是非常少见的。

实现细节

上面所说的，就是为什么我们认为数据库只是一种实现细节的原因。数据库终究只是在硬盘与内存之间相互传输数据的一种手段而已，它真的可以被认为只是一个长期存储数据的、装满字节的大桶。我们通常并不会真的以这种形式来使用数据。

因此，从系统架构的视角来看，真的不应该关心数据在旋转的磁盘表面上以什么样的格式存在。实际上，系统架构应该对磁盘本身的存在完全不关心。

但性能怎么办呢

性能难道不是系统架构的一个考量标准吗？当然是——但当问题涉及数据存储时，这方面的操作通常是被封装起来，隔离在业务逻辑之外的。也就是说，我们确实需要从数据存储中快速地存取数据，但这终究只是一个底层实现问题。我们完全可以在数据访问这一较低的层面上解决这个问题，而不需要让它与系统架构相关联。

一段轶事

在 20 世纪 80 年代末，我曾在一家创业公司中带领一组软件工程师开发和推广一个用于监控 T1 线路通信质量的网络管理系统。该系统从 T1 线路两端的设备抓取数据，然后利用预测算法来检测和汇报问题。

我们当时采用的是 UNIX 平台，并将数据存储成简单的可随机访问的格式。该项目当时也不需要用到关系型数据库，因为数据之间几乎没有内容之间的关系，用树以及链表的形式来存储数据就够了。简单来说，我们的数据存储格式是为了便于加载到内存中处理而设计的。

创业公司后来招聘了一个市场推广经理——他人很好，知识也很全面。然而他告诉我的第一件事就是我们系统中必须有一个关系型数据库。这容不得商量，也不是一个工程问题——而是一个市场问题。

这对我来说很难接受，为什么我要将链表和树重新按照表格与行模式重组，并且用SQL方式存储呢？为什么我们要在随机访问文件系统已经足够用的情况下引入大型关系型数据库系统？所以我一直和他针锋相对，互不相让。

后来公司内有一位硬件工程师被关系型数据库大潮所感染：他坚信我们的软件系统在技术上有必要采用关系型数据库。他背着我召集了公司的管理层开会，在白板上画了一间用几根杆子支撑的房子，问道：“谁会把房子建在几根杆子搭起来的地基上？”这背后的逻辑是：通过关系型数据库将数据存储于文件系统中，在某种程度上要比我们自己存储这些文件更可靠。

我当然没有放弃，一直不停地和他还有市场部斗争到底。我誓死捍卫了自己的工程原则，不停地开会、斗争。

最终，这位硬件工程师被提拔为软件开发经理，最终，系统中也加入了一个关系型数据库。最终，我不得不承认，他们是对的，而我是错的。

但这里说的不是软件工程问题：在这个问题上我仍然坚持自己没有错，在系统的核心架构中的确不应该引入关系型数据库。这里说我错了的原因，是因为我们的客户希望该系统中能有一个关系型数据库。他们其实也不知道为什么需要，因为他们自己是没有任何机会使用这个关系型数据库的。但这不是重点，问题的重点是我们的客户需要一个关系型数据库。它已经成为当时所有软件购买合同中的一个必选项。这背后毫无工程逻辑——是不理智的。但尽管它是不理智的、外行的、毫无根基的需求，但却是真实存在的。

这种需求是从哪里来的？其实是来自于当时数据库厂商非常有效的市场推广。他们说服了企业高管，他们的“数据资产”需要某种保护，数据库则提供了非常便捷的保护能力。

直到今天我们也能看到这种市场宣传，譬如“企业级”“面向服务的架构”这样的措辞大部分都是市场宣传噱头，而跟实际的工程质量无关。

回头想想，我在这个场景中应该怎么做呢？事实上，我当时应该在系统的某个角落接上一个关系型数据库，在维持系统核心数据结构的同时给关系型数据库提供一些安全的、受限的数据访问方式。但我没这么做，我辞职了，干起了咨询这一行。

本章小结

数据的组织结构，数据的模型，都是系统架构中的重要部分，但是从磁盘上存储/读取数据的机制和手段却没那么重要。关系型数据库强制我们将数据存储成表格并且以 SQL 访问，主要是为了后者。总而言之，数据本身很重要，但数据库系统仅仅是一个实现细节。

第 31 章

Web 是实现细节

20 世纪 90 年代的时候，你已经是程序员了吗？还记得 Web 是如何改变一切的吗？你记得我们在有了崭新的 Web 技术之后，是如何鄙视那些老旧的客户端/服务器架构的吗？

然而，Web 技术事实上并没有改变任何东西，或者说它也没有能力改变任何东西。这一次 Web 热潮只是软件行业从 1960 年来经历的数次振荡中的一次。这些振荡一会儿将全部计算资源集中在中央服务器上，一会儿又将计算资源分散到各个终端上。

事实上，在过去十年内，或者说自 Web 技术被普遍应用以来，这样的振荡也发生了几次。一开始我们以为计算资源应该集中在服务器集群中，浏览器应该保持简单。但随后我们又开始在浏览器中引入 Applets。再后来我们又改了主意，发明了 Web 2.0，用 Ajax 和 JavaScript 将很多计算过程挪回浏览器中。我们先是非常兴奋地将整个应用程序挪到浏览器去执行，后来又非常开心地采用 Node 技术将那些 JavaScript 代码挪回服务器上执行。

一声叹息！

无尽的钟摆

当然，这些振荡也不是从 Web 技术开始的。在 Web 出现之前，这种振荡在客户端/服务器架构中就很普遍。再往前，就是中央小型机/瘦终端的模型（这里的瘦终端和现在我们所谓的现代浏览器非常相似）。再往前则是大型计算机与打孔卡……

而且这样的故事还会继续下去，我们似乎永远也决定不了应该将计算资源放在哪里。我们不停地在集中式和分布式之间来回切换。看起来，这样的振荡还要再持续一段时间。

但从 IT 技术发展历史的整体来看，我们会发现 Web 技术的出现并没有改变任何东西。Web 技术的热潮只是在这个早于我们出生，也肯定会超过我们职业生涯的振荡周期中的一瞬间。

而且作为一名系统架构师，我们应该把眼光放长远一点，这些振荡只是短期问题，不应该把它们放在系统的核心业务逻辑中来考虑。

下面，我们来聊聊 Q 公司的故事。该公司构建了一个非常流行的个人财务系统，这是一个 GUI 很好用的桌面程序，我很喜欢它。

然后 Web 技术的热潮到来了，Q 公司打算在下一个版本中将该系统的 GUI 改成了浏览器风格。这真是犹如晴天霹雳！究竟是市场部的哪位“大神”决定要让一个桌面版的个人财务软件展示浏览器风格的呢？

我当然非常痛恨新的 UI，显然其他人也这么认为——因此在随后的几个版本里，Q 公司又逐渐将浏览器相关的设计从界面中去掉了，最终这个软件又回到正常的桌面 UI 模式。

假设你是 Q 公司的软件架构师，市场人员说服了高层管理者，要将整个 UI 重新设计为“Web”版。你应该怎么办？换句话说，在这类事情发生之前，我们应该提前做好哪方面的准备，才能应对这种无厘头的要求？

我们应该做的就是将业务规则与 UI 解耦。我不知道 Q 公司的软件架构师是否是这么做的，我也很想了解他们的故事。如果当时我在，我一定会全力游说他们将业务逻辑与 UI 解耦，因为谁知道市场推广人员接下来会想出什么好点子？

再说一下 A 公司的故事，他们的产品是智能手机。最近他们发布了一个“操作系统”的升级版（谈论一个手机的操作系统本身就够奇怪的了！）。抛去别的改动不说，这次“操作系统”的更新大幅修改了各种应用程序的外观。为什么？估计是因为市场部某位“大神”的要求吧。

我不了解这个设备中软件的细节，所以不知道这次改动是否显著影响了那些给 A 公司的手机开发应用程序的人。我只能希望 A 公司的系统架构师，以及应用程序的系统架构师能将 UI 和业务逻辑分离，因为这些市场推广人员是不会错过这里任何一丁点儿的耦合关系的。

总结一下

将上面的故事总结成一句话，就是：GUI 只是一个实现细节。而 Web 则是 GUI 的一种，所以也是一个实现细节。作为一名软件架构师，我们需要将这类细节与核心业务逻辑隔离开来。

其实我们可以这样考虑这个问题：Web 只是一种 I/O 设备。早在 20 世纪 60 年代，我们就已经了解编写设备无关应用程序的重要性。这种独立性的重要性至今仍然没有变化，Web 也不例外。

是这样的吗？有人可能会辩称 Web 这样的 GUI 是非常特殊的，它能力强大，强大到让我们追求设备无关的架构变得毫无意义。当我们考虑到 JavaScript 数据校验的复杂程度、可拖拽的 Ajax 调用，以及无数可以轻松引入的设计组件时，很容易认为追求设备无关性是不现实的。

从某种程度上来说，的确如此。应用程序和 GUI 之间的频繁交互的确是与 GUI 的类型密切相关的。浏览器与 Web 应用之间的交互模式也的确与桌面客户端/服务器之间的交互模式区别很大。想要让浏览器上的 Web 操作模仿我们在 UNIX 中对 I/O 设备那样的操作，将其抽象成界面交互模型几乎是不可能的。

但我们其实可以从 UI 和应用程序之间的另一条边界出发来进行抽象化。因为业务逻辑可以被视为是一组用例的集合。而每个用例都是以用户的身份来执行某种操作的，所以它们都可以用输入数据、处理过程以及输出数据这个流程来描述。

也就是说，在 UI 和应用程序之间的某一点上，输入数据会被认为达到了一个完整状态，然后用例就被允许进入执行阶段了。在用例执行完之后，其生成的返回数据又继续在 UI 与应用程序之间传递。

这样一来，完整的输入数据，以及完整的输出数据就可以被标准化为数据结构，并提供给执行用例的进程了。通过这种方法，我们就可以认为用例都是以设备无关的方式在操作 I/O 设备。

本章小结

这种抽象化处理并不容易，很有可能需要经历几个来回才能找到正确的方向，但这是完全可行的。由于世界上最不缺少的就是市场“大神”，很多时候做这些事情还真的是非常有必要的。

第 32 章

应用程序框架是实现细节

应用程序框架现在非常流行，这在通常情况下是一件好事。许多框架都非常有效，非常有用，而且是免费的。

但框架并不等同于系统架构——尽管有些框架确实以此为目标。

框架作者

大部分框架的作者愿意免费提供自己的工作成果，是因为他们想要帮助整个社群，想要回馈社会。这值得鼓励，但不管这些作者的动机有多么高尚，恐怕也并没有提供针对你个人的最佳方案。即使他们想，也做不到，因为他们并不了解你，也不了解你遇到的问题。

这些框架作者所了解的都是他们自己遇到的问题，可能还包括亲戚朋友所遇到的。他们创造框架的目的是解决这些问题——而不是解决你遇到的问题。

当然，你所遇到的问题可能和其他人遇到的大体上一致。如果不是这样，框架也就不会那么流行了。正是由于这种重合性的存在，框架才这么有用。

单向婚姻

我们与框架作者之间的关系是非常不对等的。我们要采用某个框架就意味着自己要遵守一大堆约定，但框架作者却完全不需要为我们遵守什么约定。

请仔细想想这一关系，当我们决定采用一个框架时，就需要完整地阅读框架作者提供的文档。在这个文档中，框架作者和框架其他用户对我们提出进行应用整合的一些建议。一般来说，这些建议就是在要求我们围绕着该框架来设计自己的系统架构。譬如，框架作者会建议我们基于框架中的基类来创建一些派生类，并在业务对象中引入一些框架的工具。框架作者还会不停地催促我们将应用与框架结合得越紧密越好。

对框架作者来说，应用程序与自己的框架耦合是没有风险的。毕竟作为作者，

他们对框架有绝对的控制权，强耦合是应该的。

与此同时，作者当然是非常希望让我们的应用与其框架紧密结合的，因为这意味着脱离框架会很困难。作为框架作者来说，没有什么比让一堆用户心甘情愿地基于他的框架基类来构建派生类更自豪的事情了。

换句话说，框架作者想让我们与框架订终身——这相当于我们要对他们的框架做一个巨大而长期的承诺，而在任何情况下框架作者都不会对我们做出同样的承诺。这种婚姻是单向的。我们要承担所有的风险，而框架作者则没有任何风险。

风险

那么我们要承担的风险究竟有哪些呢？我们可以想到的至少有以下这几项：

- 框架自身的架构设计很多时候并不是特别正确的。框架本身可能经常违反依赖关系原则。譬如，框架可能会要求我们将代码引入到业务对象中——甚至是业务实体中。框架可能会想要我们将框架耦合在最内圈代码中。而我们一旦引入，就再也不会离开该框架了，这就像戴上结婚戒指一样，从此一生不离不弃了。
- 框架可能会帮助我们实现一些应用程序的早期功能，但随着产品的成熟，功能要求很可能超出框架所能提供的范围。而且随着时间的推移，我们也会发现在应用的开发过程中，自己与框架斗争的时间要比框架帮助我们的时间长得多。
- 框架本身可能朝着我们不需要的方向演进。也许我们会被迫升级到一个并不需要的新版本，甚至会发现自己之前所使用的旧功能突然消失了，或悄悄改变了行为。
- 未来我们可能会想要切换到一个更新、更好的框架上。

解决方案

解决方案是什么呢？

请不要嫁给框架！

我们可以使用框架——但要时刻警惕，别被它拖住。我们应该将框架作为架构最外圈的一个实现细节来使用，不要让它们进入内圈。

如果框架要求我们根据它们的基类来创建派生类，就请不要这样做！我们可以创造一些代理类，同时把这些代理类当作业务逻辑的插件来管理。

另外，不要让框架污染我们的核心代码，应该依据依赖关系原则，将它们当作核心代码的插件来管理。

以 Spring 为例，它作为一个依赖注入框架是不错的，也许我们会需要用 Spring 来自动连接应用程序中的各种依赖关系。这不要紧，但是千万别在业务对象里到处写`@autowired`注解。业务对象应该对 Spring 完全不知情才对。

反之，我们也可以利用 Spring 将依赖关系注入到 `Main` 组件中，毕竟 `Main` 组件作为系统架构中最低层、依赖最多的组件，它依赖于 Spring 并不是问题。

不得不接受的依赖

有一些框架是避免不了使用的。例如，如果你在用 C++，那么 STL 就是很难避免使用的。如果你在用 Java，那么标准类库也是不太可能避免使用的。

这很正常——但这仍然应该是你主动选择的结果。你必须明白，如果一旦在项目中引入一个框架，很有可能在整个生命周期中都要依赖于它，不管后来情形怎么变化，这个决定都很难更改了。因此，不应该草率地做出决定。

本章小结

总而言之。当我们面临框架选择时，尽量不要草率地做出决定。在全身心投入之前，应该首先看看是否可以部分地采用以增加了解。另外，请尽可能长时间地将框架留在架构边界之外，越久越好。因为谁知道呢，也许你可以不用买奶牛也能喝到牛奶[1]。

1 美国俚语，意思是既然可以直接达到目的就不必绕弯路。——译者注

第 33 章

案例分析：视频销售网站

现在是时候将所有的这些设计规则和架构理念整合起来了。下面，我们来做一次案例分析，这个案例分析虽然很简短，但可以描述清楚一个优秀的系统架构师在设计过程和设计决策中应该如何行事。

产品

在这个案例分析中，我要讲的是一个我自己很熟悉的产品：线上收费视频网站。当然，这个有点像 cleancoders.com，我在这个网站上出售我的软件开发教程视频。

这个案例的设计很简单，就是我们打算向一些个人或者企业提供一批收费的线上教学视频。个人用户既可以选择在线支付之后直接在线观看视频，也可以选择付一笔更高的费用将视频下载到本地，永久地拥有它们。而企业用户就只能在线播放，但他们可以选择批量购买，以此来获得一定折扣。

个人用户通常既是购买者又是观看者。而企业用户则不同，他们购买视频通常是用来给其他人观看的。

视频作者需要负责上传视频文件、写简介，并且提供视频附带的一系列习题、课后作业、答案、源代码以及其他各类资料。

管理员需要负责增加新的视频播放列表，往视频播放列表里添加和删除视频，并且为各种许可类型设置价格。

系统架构设计中的第一步，是识别系统中的各种角色和用例。

用例分析

下面，我们通过图 33.1 来示范一次典型的用例分析。

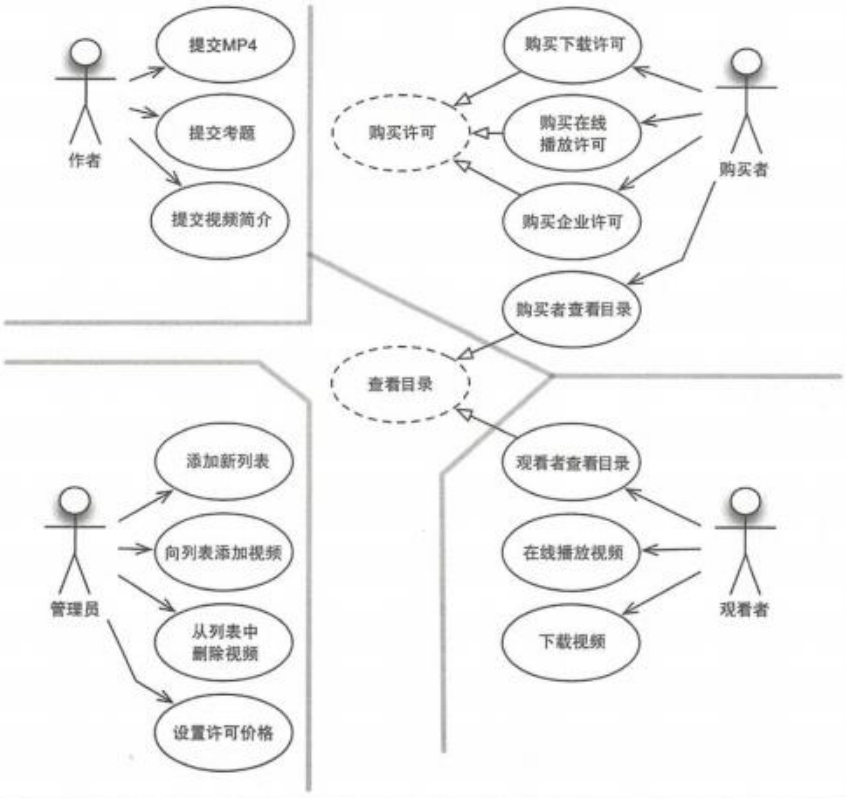


图 33.1：典型的用例分析

如你所见，图中显然存在着四个角色。根据单一职责原则（SRP），这四个角色将成为系统变更的主要驱动力。每当添加新功能，或者修改现有功能时，我们所做的一切都是在为这些角色服务。所以我们希望能够对系统进行分区处理，避免其中一个角色的变更需求影响其他角色。

另外，图 33.1 中的用例并不是一个完整的列表。例如，这里没有分析用于执行登录、注销的用例。省略它们的原因很简单，为了控制本书篇幅。如果列出所有的用例，这一章就会变成一本单独的书了。

读者应该注意到图 33.1 中还有一些用虚线框起来的用例。我们称之为抽象用

例[1]，它们通常用来负责设置通用策略，然后交由其他具体用例来使用。譬如在该图中，“查看目录”这个用例同时被“购买者查看目录”和“观看者查看目录”这两个用例所继承并实现。

一方面来说，其实这种抽象并不是必需的。如果没有这一层抽象，整个产品并不会受到影响；但是另一方面来说，由于这两个用例十分相近，我认为以某种方式来将它们合并起来分析是很合理的。

组件架构

既然我们弄清楚了系统中的各种角色和用例，接下来就可以构造一个初步的组件架构图了（如图 33.2 所示）。

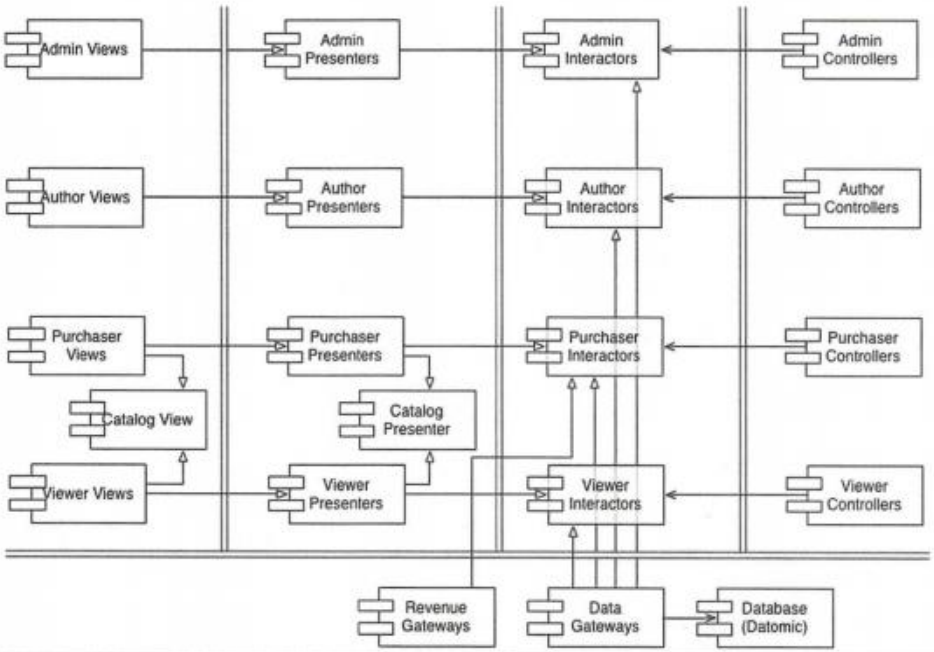


图 33.2：初步的组件架构图

1 这里的“抽象”是我自己定义的。严格来讲，使用 UML 的<<abstract>>这种原型符号更标准一些，但我觉得如今再遵守这些刻板套路已经没有什么意义了。

在该图中，双实线代表了系统架构边界。可以看到这里将系统划分成视图、展示器、交互器以及控制器这几个组件，同时也按照对应的系统角色进行了分组。

图 33.2 中的每一个组件都对应着一个潜在的.jar 文件或.dll 文件。每一个组件都会包含归属于它的视图、展示器、交互器、控制器文件。

值得注意的是，这里有两个特殊的组件：目录视图（Catalog View）和目录展示器（Catalog Presenter）。这就是我应对查看目录列表这个抽象用例的方法。我假设这些视图和展示器将会被编写为抽象类，而继承它们的组件将会包括它们的派生类。

但问题是，我们真的需要将系统拆分成这么多组件，然后以.jar 或.dll 文件的形式一个个交付吗？是，又不全是。我们确实要按照组件将编译和构建环境分开，以便单独构建对应的组件。但我们仍然可以考虑将所有的交付单元组合起来交付。例如，根据图 33.2 中的分组，我们可以很简单地将它们交付为 5 个.jar 文件——视图、展示器、交互器、控制器和工具类，这样就可以分别单独部署这些被修改的组件了。

除此之外，还有另一种分组方式，就是将视图和展示器放在同一个.jar 文件中，而将交互器、控制器以及工具类各自放在独立的.jar 文件中。还有一种更简单的方式，就是将视图和展示器放在一个.jar 文件中，而将其他所有的组件合并为另一个.jar 文件。

随着系统的演进，我们可以根据系统变更来调整部署方式。

依赖关系管理

如你所见，图 33.2 中的控制流是从右向左的。输入发生在控制器端，然后输入的数据经交互器处理后交由展示器格式化出结果，最后由视图来展示这个结果。

请注意，图中的箭头并不是一直从右向左的。事实上大部分的箭头都是从左向右的。这是因为该架构设计要遵守依赖关系原则。所有跨越边界的依赖关系都应该

是同一个方向，而且都指向包含更高级策略的组件。

另外，还应该注意一下图中的“使用”关系（开放箭头），它和控制流方向是一致的；而“继承”关系（闭合箭头）则与之相反，它反映的是我们对开闭原则的应用，通过调整依赖关系，可以保证底层细节的变更不会影响到高层策略组件。

本章小结

图 33.2 中的架构实现的是两个维度上的隔离。第一个是根据单一职责原则对所使用的系统的各个角色进行了隔离，第二个则是对依赖关系原则的应用。这两个维度的隔离都是为了将不同变更原因和不同变更速率的组件分隔开来。譬如变更的原因不同是因为组件使用的角色不同，而变更速率则取决于组件所在的层级。

按照这样的方式组织代码的结构，我们就可以在部署时做灵活的选择。可以随时将组件整合部署，也可以在要求变化的时候灵活地调整。

第 34 章

拾遗

本章由 Simon Brown 撰写

根据本书之前给出的所有建议，相信读者一定能够建构出具有良好边界设计的类和组件，以形成清晰的责任划分以及可控的依赖关系，设计出更好的软件了。但是困难之处往往在于细节之中，一旦疏忽，也有可能会对软件质量造成不良影响。

下面我们再来看一个例子，假设正在构建一个在线书店，这个例子的任务是实现一个客户查看订单状态的用例。虽然这是一个 Java 程序的示例，但其所示范的原理适用于任何语言。现在，让我们暂时将整洁架构的概念放在一边，先来看一下如何具体安排代码设计和代码结构。

按层封装

我们首先想到的，也可能是最简单的设计方式，就是传统的水平分层架构。在这个架构里，我们将代码从技术角度进行分类。这通常被称为“按层封装”。图 34.1 用 UML 类图展示了这种设计。

在这种常见的分层架构中，Web 代码分为一层，业务逻辑分为一层，持久化是另外一层。换句话说，我们对代码进行了水平分层，相同类型的代码在一层。在“严格的分层架构”中，每一层只能对相邻的下层有依赖关系。在 Java 中，分层的概念通常是用包来表示的。如图 34.1 所示，所有的分层（包）之间的依赖关系都是指向下的。这里包括了以下 Java 类。

- `OrdersController`：Web 控制器，类似 Spring MVC 控制器，负责处理 Web 请求。
- `OrderService`：定义订单相关业务逻辑的接口。
- `OrderServiceImpl`：Order 服务的具体实现[1]。
- `OrdersRepository`：定义如何访问订单持久信息的接口。
- `JdbcOrderRepository`：持久信息访问接口的实现。

1 这其实是一种特别糟糕的命名方式，但是如同下文所说，这真的不算什么。

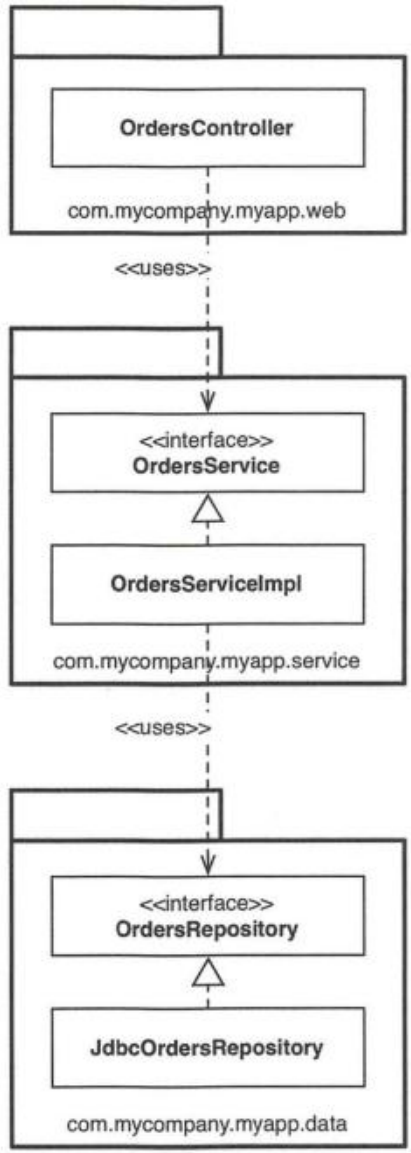


图 34.1：按层封装

在 *Presentation Domain Data Layering* 这篇文章中[1]，Martin Fowler 声称采用这种分层架构是初期的一个不错选择。他的观点并不缺乏拥戴者。很多书籍、教程和代

1　网址为 https://martinfowler.com/bliki/PresentationDomainDataLayering.html。

码示范都在教育你采用分层架构。

这种方式在项目初期之所以会很合适，是因为它不会过于复杂。但就像 Martin 指出的那样，一旦软件规模扩展了，我们很快就会发现将代码分为三大块并不够，需要进一步进行模块化。

如 Bob 所说，这里还存在另外一个问题是，分层架构无法展现具体的业务领域信息。把两个不同业务领域的、但是都采用了分层架构的代码进行对比，你会发现它们的相似程度极高：都有 Web 层、服务层和数据仓库层。这是分层架构的另外一个问题，后文会具体讲述。

按功能封装

另外一种组织代码的形式是“按功能封装”，即垂直切分，根据相关的功能、业务概念或者聚合根（领域驱动设计原则中的术语）来切分。在常见的实现中，所有的类型都会放在一个相同的包中，以业务概念来命名。

图 34.2 展示了这种方式，类和接口与之前类似，但是相比之前，这次它们都被放到了同一个 Java 包中。相比“按层封装”，这只是一个小变化，但是现在顶层代码结构至少与业务领域有点相关了。我们可以看到这段代码是与订单有关的，而不是只能看到 Web、服务及数据访问。另外一个好处是，如果需要修改“查看订单”这个业务用例，比较容易找到相关代码，毕竟它们都在一个包中，而不是分散在各处。[1]

软件研发团队常常一开始采用水平分层方式（即“按层封装”），遇到困难后再切换到垂直分层方式（即“按功能封装”）。我认为，两种方式都很不好。看完本书，你应该意识到还有更好的分类方式——没错。

1 由于现代 IDE 的导航功能日益增强，这方面的好处已经不太明显了。但目前有一股力量在推动大家重返轻量级的文本编辑器，这背后的原因我这个老古董就不太能理解了。

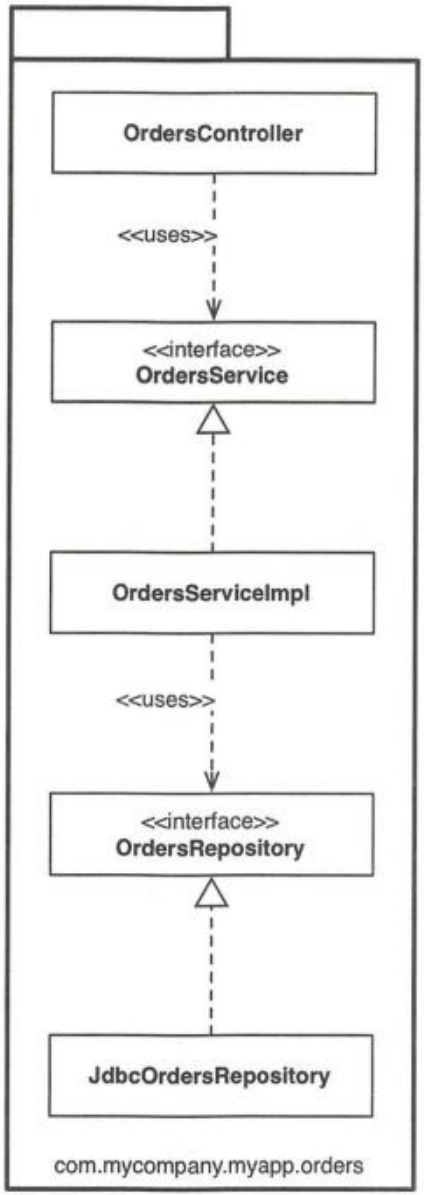

OrdersController
<<uses>>
<<interface>>
OrdersService
OrdersServiceImpl
<<uses>>
<<interface>>
OrdersRepository
JdbcOrdersRepository
com.mycompany.myapp.orders

图 34.2：按功能封装

端口和适配器

如 Bob 大叔所说，通过采用“端口和适配器”“六边形架构”“边界、控制器、实体”等，我们可以创造出一个业务领域代码与具体实现细节（数据库、框架等）隔离的架构。总结下来，如图 34.3 所示，我们可以区分出代码中的内部代码（领域，Domain）与外部代码（基础设施，Infrastructure）。

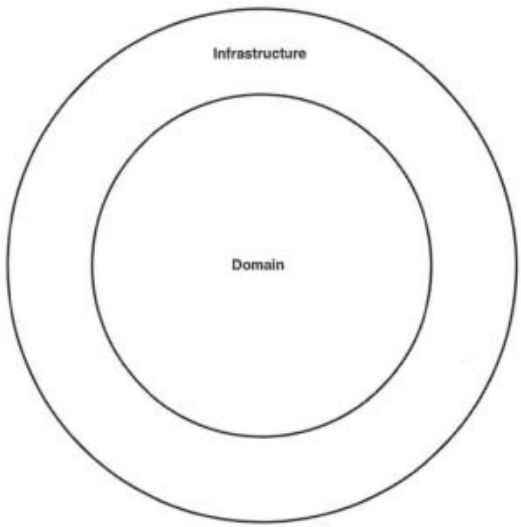


图 34.3：区分内部代码和外部代码

内部区域包含了所有的领域概念，而外部区域则包含了与外界交互的部分（例如 UI、数据库、第三方集成等）。这里主要的规则是，只有外部代码能依赖内部代码，反之则不能。图 34.4 展示了“查看订单”这个业务用例是如何用这种方式实现的。

这里 `com.mycompnay.myapp.domain` 包是内部代码，另外一个包是外部代码。注意这里的依赖关系是由外向内的。眼尖的读者可以注意到之前的 `OrderRepository` 类现在被改名为 `Orders`。这个概念基于领域驱动设计理念，其中要求内部代码都应该用独特的领域语言来描述。换句话说，我们在业务领域里面讨论的应该是“Orders”，而不是“OrdersRepository”。

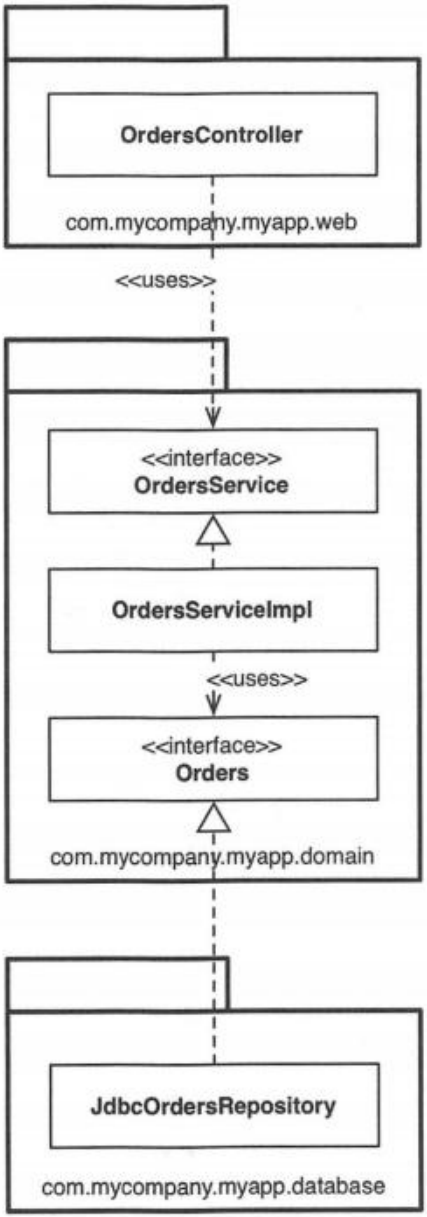


图 34.4：“查看订单”业务用例

值得注意的是，这里是 UML 类图的一个简化版，这里缺少了交互器，以及跨边界调用时对应的数据编码解码对象。

按组件封装

虽然我对本书中的 SOLID、REP、CCP、CRP 以及其他大部分建议完全认同，我想提出对代码组织方式的一个不同看法——“按组件封装”。一些背景信息：在我的职业生涯中，我基于 Java 构建了大量不同领域的企业软件，这些软件系统要求各异。大部分系统都是基于 Web 的，也有一些是 CS 架构[1]，或者是分布式架构的、基于消息的，或者其他的。虽然具体采用的技术不同，但大部分系统都是基于传统的分层架构的。

我已经给出一些分层架构不好的理由，但这还不是全部。分层架构设计的目的是将功能相似的代码进行分组。处理 Web 的代码应该与处理业务逻辑的代码分开，同时也与处理数据访问的代码分开。正如我们在 UML 类图中所见，从实现角度讲，层就是代表了 Java 包。从代码可访问性角度来讲，如果需要 `OrdersController` 依赖 `OrderService` 接口，那么这个接口必须设置为 `public`，因为它们在不同的包中。同样的，`OrdersRepository` 接口也需要设置为 `public`，这样才能被包外的类 `OrdersServiceImple` 使用。

在严格分层的架构中，依赖指向的箭头应该永远向下，每一层只能依赖相邻的下一层。通过引入一些代码互相依赖的规则，我们就形成了一个干净、漂亮的单向依赖图。这里有一个大问题——只要通过引入一些不应该有的依赖来作弊，依然可以形成漂亮的单向依赖图。

假设新员工加入了团队，你给新人安排了一个订单相关的业务用例的实现任务。由于这个人刚刚入职，他想好好表现，尽快完成这项功能。粗略看过代码之后，新人发现了 `OrdersController` 这个类，于是他将新的订单相关的 Web 代码都塞了进去。但是这段代码需要从数据库查找一些订单数据。这时候这个新人灵机一动：

1 我 1996 年大学毕业之后，第一份工作就是用 PowerBuilder 构造一个客户端-服务器架构的桌面应用。PowerBuilder 是一个超级好用的 4GL 组件，十分适合构造基于数据库的应用。而几年之后，在另外一个项目里，我需要自己创建数据库链接器（这时候 JDBC 还没有被发明），以及用 AWT 构造自己的 UI 组件。这就是所谓的时代进步！

“代码已经有了一个 `OrdersRepository` 接口，只需要将它用依赖注入框架引入控制器就行，我真机智！”几分钟之后，功能已经正常了，但是 UML 结构图变成了图 34.5 这样。

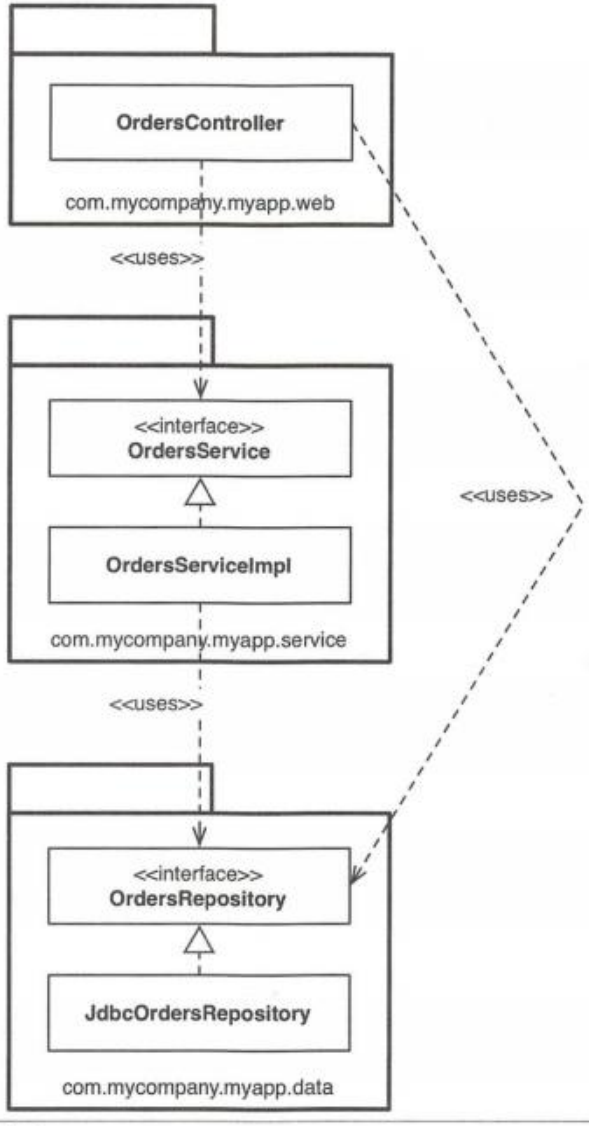


图 34.5：宽松的分层架构

依赖关系箭头依然向下，但是现在 `OrdersController` 在某些情况下绕过了 `OrderService` 类。这种组织形式被称为*宽松的分层架构*，允许某些层跳过直接相邻的邻居。在有些情况下，这是意料之中的——例如，如果我们在遵循 CQRS 设计模式[1]，这是合理的。但是更多的情况下，绕过业务逻辑层是不合理的，尤其是在业务逻辑层要控制权限的情况下。

虽然新的业务用例可以正常工作，但是它可能不是按照合理方式实现的。作为咨询师，我曾经见过很多团队出现这种情况，只有他们开始仔细观察自己的代码结构图时才会发现。

这里我们有的其实只是一个规范——一个架构设计原则——内容是“Web 控制器永远不应该直接访问数据层”。这里的核心问题当然是如何强制执行。我遇见的很多团队仅仅通过采用“自律”或者“代码评审”方式来执行，“我相信我的程序员”。有这种自信当然很好，但是我们都知道当预算缩减、工期临近的时候会发生什么事情。

有一小部分团队告诉我，他们会采用静态分析工具（例如 Ndepend、Structure101、Checkstyle）来在构建阶段自动检查违反架构设计规则的代码。估计你见过这种代码，一般来说就是一段正则表达式，例如“包`**/web` 下面的类型不允许访问`**/data` 下面的类型”，这些检查在编译步骤之后执行。

这种方式虽然简单粗暴，但是确实能起效果，可以锁定违反了团队定义的系统架构设计原则的情况，并且（理想情况下）导致构建失败。这两种方法的共同问题是容易出错，同时反馈循环时间太长了。如果不精心维护，整个代码库可能很快就变成“一团泥巴”[2]。我个人更倾向选择能够让编译器执法的做法。

那么，看一下“按组件封装”的做法。这种做法混合了我们之前讲的所有的方法，目标是将一个粗粒度组件相关的所有类放入一个 Java 包中。这就像是以一种面向服务的视角来构建软件系统，与微服务架构类似。这里，就像端口和适配器模式将 Web 视为一种交付手段一样，“按组件封装”将 UI 与粗粒度组件分离。图 34.6

1 在命令查询责任分离设计模式中，更新和读取数据的模式是不同的。

2 参见 http://www.laputan.org/mud/。

展示了“查看订单”这个用例的设计图。

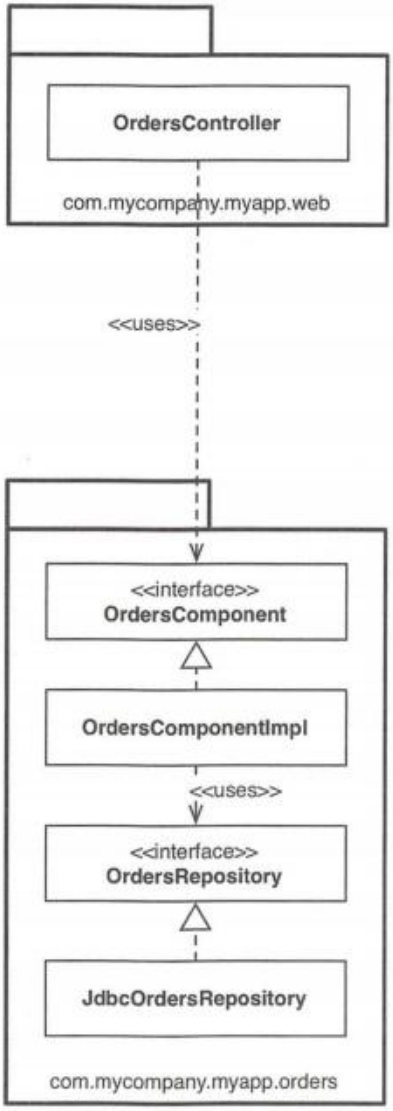


图 34.6：“查看订单”业务用例

总的来说，这种方式将“业务逻辑”与“持久化代码”合并在一起，称为“组件”，Bob 大叔在本书中对“组件”的定义如下：

> 组件是部署单元。组件是系统中能够部署的最小单位，对应在 Java 里就是 jar 文件。

我对组件的定义稍有不同："在一个执行环境（应用程序）中的、一个干净、良好的接口背后的一系列相关功能的集合"。这个定义来自我的"C4 软件架构模型"[1]。这个模型以一种层级模型讨论软件系统的静态结构，其中的概念包括容器、组件、类。这个模型认为，系统由一个或者多个容器组成（例如 Web 应用、移动 App、独立应用、数据库、文件系统），每个容器包含一个或多个组件，每个组件由一个或多个类组成。每个组件具体存在于哪个 jar 文件中则是另外一个维度的事情。

这种"按组件封装"的方式的一个好处是，如果我们需要编写和订单有关的代码，只有一个位置需要修改——`OrdersComponet`。在这个组件中，仍然应该关注重点隔离原则，但这是组件内部问题，使用者不需要关心。这就有点像采用微服务架构，或者是面向服务架构的结果——独立的 `OrderService` 会将所有订单相关的东西封装起来。这里关键的区别是解耦的方式。我们可以认为，单体程序中的一个良好定义的组件，是微服务化架构的一个前提条件。

具体实现细节中的陷阱

表面上看，四种代码组织方式各不相同，可以认为是不同的架构设计风格。可是，如果具体实现中不严加注意，很快就会出现偏差。

我经常遇到的一个问题是，Java 中 `public` 访问控制修饰符的滥用。我们作为程序员，好像天生就喜欢使用 `public` 关键词。这就好像是肌肉记忆一样。如果不信，请看一下各种书籍的代码示范、各种入门教程，以及 GitHub 上的开源框架。这个趋势是显而易见的，不管采用了哪种系统架构风格。

将所有的类都设置为 `public` 意味着就无法利用编程语言提供的封装手段。这样一来，没有任何东西可以阻碍某人写一段直接初始化具体实现类的代码，哪怕它违反了架构设计的要求。

1 参见 https://www.structurizr.com/help/c4。

组织形式与封装的区别

从另外一个角度来看，如果我们将 Java 程序中的所有类型都设置为 public，那么包就仅仅是一种组织形式了（类似文件夹一样的分组方式），而不是一种封装方式。由于 public 类型可以在代码库的任何位置调用，我们事实上就可以忽略包的概念，因为它并不提供什么价值。最终，如果忽视包的概念（因为并不起到任何封装和隐藏的功能），那么想要采用的任何架构风格就都不重要了。我们回过头来看一下例子中的 UML 图，如果所有的类型都是 public，那么 Java 包就成了一个无关紧要的细节信息。于是，所有四种架构方式事实上并没有任何区别（参见图 34.7）。

我们再详细看一下图 34.7 中各个类之间的箭头：不论采用哪种架构设计风格，它们的指向都是一致的。虽然概念不同，但是语法上都是一致的。更进一步说，如果所有的类都是 public 的，那么其实我们就是在用四种不同的方式描述一个传统的分层架构设计方式。你会说当然没有人会将所有的 Java 类都设置为 public，但是相信我，我见过。

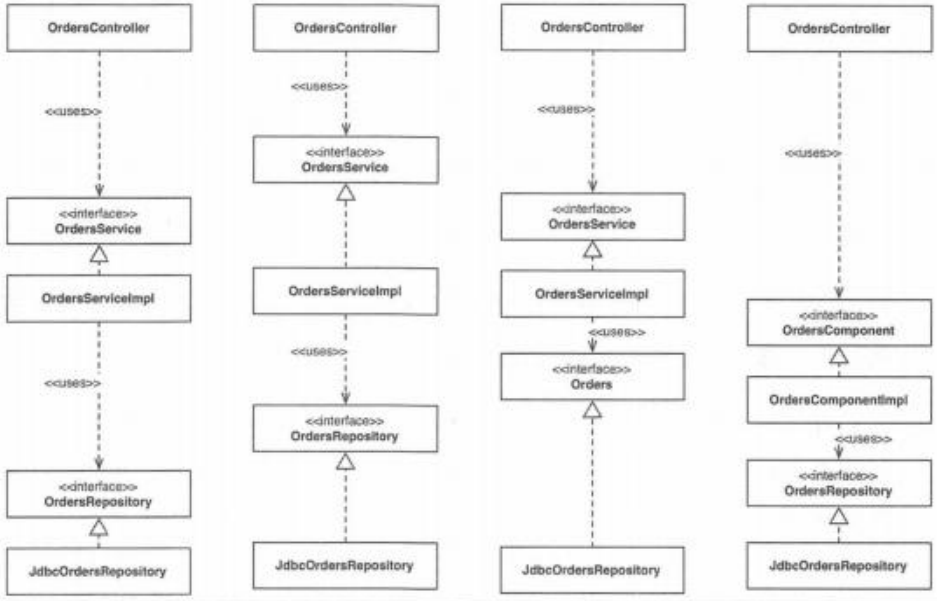


图 34.7：四种系统架构设计风格其实是等同的

虽然Java中的访问修饰符并不完美[1]，但是忽略它们的存在就是在自找麻烦。Java类与包的组织形式其实可以很大程度决定这个类的可访问性（或者不可访问性）。如果我们将包的概念引入这幅图，同时标记（虚化的形式展示）应用到访问控制符的地方，这个图就很有意思了（参见图34.8）。

从左向右，在“按层封装”方式中，`OrderService` 与 `OrderRepository` 需要 `public` 修饰符，因为包外的类需要依赖它们。然而，具体实现类（`OrderServiceImpl` 和 `JdbcOrdersRepository`）则可以设置更细致的访问权限（包范围内的protected）。不需要有人依赖它们，它们是具体的实现细节。

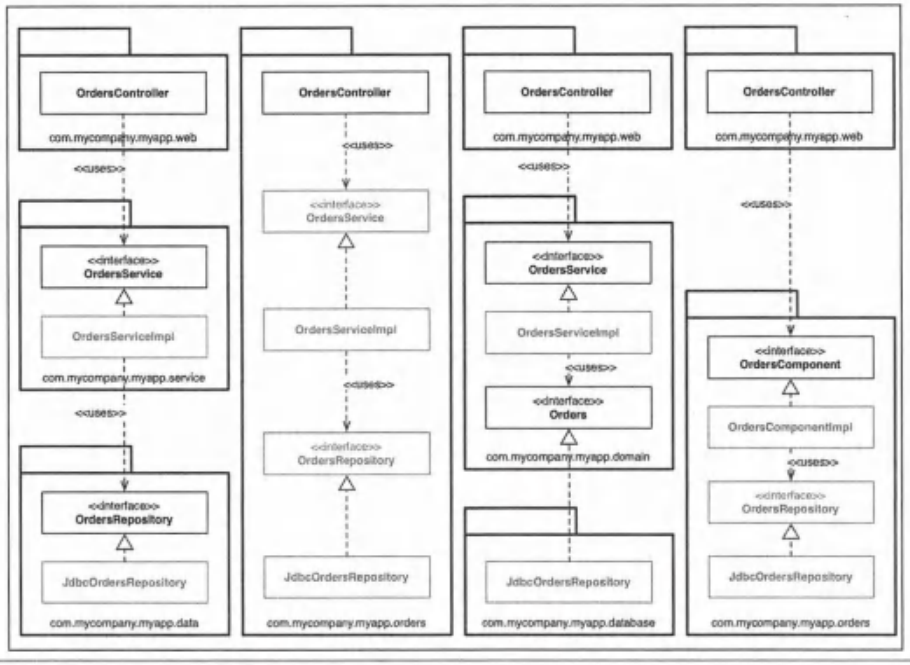


图34.8：带有访问修饰符的类型被虚化了

在“按功能封装”模式中，`OrdersController` 是整个包的入口，所以其他的类都可以设置为包范围内的protected。这里的一个问题是，代码库中的其他代码都必须通过控制器才能访问订单信息——这可能是好处，也可能是坏处，视实际情

1 例如在Java中，虽然我们倾向于认为包是具有层级关系的，但是其实我们并不能控制包和子包之间的访问关系。任何层级关系仅仅在磁盘目录结构上有意义。

况而定。

在端口与适配器模式中，`OrderService` 与 `Orders` 接口都有来自包外的依赖关系，所以需要 `public` 修饰符。同样，实现类可以设置为包范围内 protected，依赖在运行时注入。

最后，在“组件”封装模式中，`OrdersComponet` 接口有来自 Controller 的依赖关系，但是其他类都可以设置为包 protected。`Public` 类型越少，潜在的依赖关系就越少。现在包外代码就不能再直接使用 `OrdersRepository` 接口或者其对应的实现[1]，我们就可以利用编译器来维护架构设计原则了。在.Net 语言中，我们可以用 `internal` 关键词达到一样的目的，然而我们需要给每个组件创建一个单独的 assembly。

再澄清一点，这里描述的全都和单体程序有关，所有代码都存放在同一个代码树下。如果你在构建这种程序（大部分程序都是如此），那么我强烈建议利用编译器来维护架构设计原理，而不要依赖个人自律和编译过程之后的工具。

其他的解耦合模式

除编程语言自带的工具之外，通常还有其他方式可以进一步解耦源代码级别的依赖关系。在 Java 语言中，有模块化框架 OSGi，以及最新的 Java 9 模块系统。正确利用模块系统，我们可以进一步区分 `public` 类型和对外发布的类型。例如，我们可以创建一个 `Orders` 模块，将所有的类型标记为 `public`，但仅仅公布一小部分类供外部调用。虽然耗时很久，但是我十分期待 Java 9 的模块系统，它能提供构建更好软件的另一套工具，希望能够再次点燃人们思考设计的热情。

另外一个选择是将代码分散到不同的代码树中，以从源代码级别解耦依赖关系。以端口和适配器方式为例，我们会有三个代码树：

1　除非你用 Java 的反射机制作弊——请别这么干！

- 业务代码（所有技术和框架无关的代码）：`OrdersService`、`OrderServiceImpl` 以及 `Orders`。
- Web 源代码：`OrdersController`。
- 持久化源代码：`JdbcOrdersRepository`。

后面两个源代码树对业务代码有编译期依赖关系，而业务代码则对 Web 和数据持久毫无所知。从实现角度来看，我们可以通过将这些代码在构建工具中组织成不同的模块或者项目（例如 Maven、Gradle、MSBUILD 等）来达到目的。理想情况下，我们可以用这种模式将所有组件都划分成不同的项目。

然而，这有点太理想化了，因为拆分代码库经常会带来性能、复杂度和维护性方面的问题。

有些人采用一个稍微简单的组织方式，仅使用两个代码树：

- 业务（Domain）代码（内部）
- 基础设施（Infrastructure）代码（外部）

这与图 34.9 完美对应，很多人都用这个方式来简化对端口和适配器架构的描述。基础设施部分对业务代码有一个编译期的依赖关系。

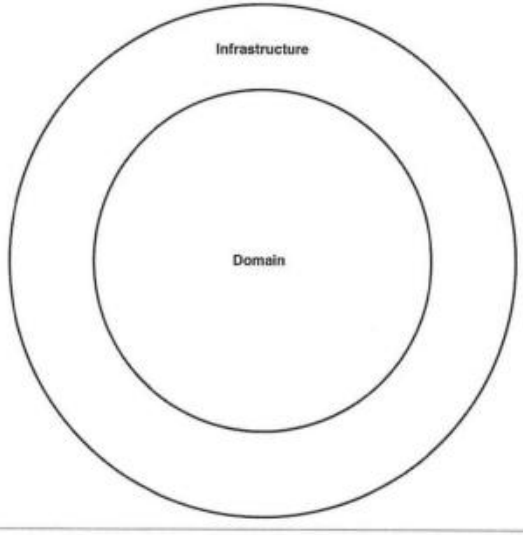


图 34.9：业务与基础设施代码

这种代码组织方式是可行的，但是需要额外注意随之而来的问题。我称这个问题为“端口与适配器模式中的 Périphérique 反模式”。法国巴黎有一条环形公路，名字是 Périphérique 大道。这条大道允许车辆环绕巴黎而不需要进入市区。同样的，将所有的基础设施代码放在同一个源代码树中，就有可能使得应用中一个区域的基础设施代码（Web 控制器）直接调用另外一个区域的代码（数据库访问），而不经过领域代码。如果没有设置正确的访问修饰符，就更是如此了。

本章小结：本书拾遗

这一章的中心思想就是，如果不考虑具体实现细节，再好的设计也无法长久。必须要将设计映射到对应的代码结构上，考虑如何组织代码树，以及在编译期和运行期采用哪种解耦合的模式。保持开放，但是一定要务实，同时要考虑到团队的大小、技术水平，以及对应的时间和预算限制。最好能利用编译器来维护所选的系统架构设计风格，小心防范来自其他地方的耦合模式，例如数据结构。所有的实现细节都是关键的！

后　序

我的软件工程师生涯开始于 20 世纪 90 年代，那是一个恐龙级大型架构统治世界的时代。要想在那样的时代获得一席之地，我们必须学会对象及其组件、设计模式、统一建模语言（包括其前身）的相关知识。

现在想起来，或许真的可以考虑把我们那段日子所做的事情叫作“童子军项目”。每个项目的开头都会有一段长长的设计阶段，以便等那些“高级”程序员为一些跟随他们的、较“低级”的程序员制订好系统的设计蓝图，当然，这些“高级”程序员似乎永远完不成这件事。

于是乎，做这件事的人被升级到了“软件架构师”，接着是“首席架构师”“总架构师”“枢密院首席架构师”以及其他各种高不可言的头衔，最终，我们还是让一切回到了原点。而我似乎注定要把时间花在画那些带箭头的盒子和编写 PowerPoint 的事情上，而这些事对真实代码的影响近乎为零。

这让我无比受挫，每一行代码本身都至少包含了一条设计决策，任何一个写代码的家伙对软件质量的影响都远在我这个 PowerPoint 专业户之上。

幸运的是，接下来发生的敏捷软件开发革命终于让我们这些架构师脱离了苦海。毕竟我是一名程序员，喜欢的是编程。而且我也发现影响软件质量最好的方法还是

编写代码。

这些大型架构像恐龙一样在大进程式的原始平原上游荡，然后被一颗叫作“敏捷开发”的小行星灭绝了，真是老天开眼啊！

现在，开发团队们可以自由地专注于真正重要的内容，并思考如何为他们所做的事情添加更多价值了。也就是说，他们现在再也不需要浪费几周或几个月的时间等待那些大型架构的设计文档了，他们可以名正言顺地忽略这些设计，直接按照自己的想法编写代码。然后，开发团队只需要安排客户直接参与测试，并在快速设计会议上得到用户的支持，然后他们就可以继续写代码了。

大型架构像恐龙一样消失了，前期设计够用、后期进行大量重构的设计思想如小巧玲珑的哺乳动物一样代替了它们，软件架构迎来了响应式设计的时代。

好吧，无论如何，以上这些都属于理论。

把架构设计工作交给程序员的问题就是，程序员必须学会像架构师一样思考问题。事实证明，我们在大型架构时代学到的东西也并非一文不值。其设计软件结构的方法依然在我们保持软件的适应和扩展能力方面有着深远的影响，即使在短期开发中也是如此。

我们的每一项设计决策都必须为未来的变化敞开大门。就像打台球一样，我们的每一杆击球都不只是为了要把球打进洞里，它也事关下一杆击球时所在的位置。让我们现在编写的代码不对未来的代码产生阻碍是一项非常重要的技能，通常需要花费多年的时间才能掌握。

因此，在大型架构时代让位给易碎型架构（Fragile Architecture）的新时代之后，虽然设计创造的价值得到了快速发展，但这也让我们想要持续创新变得举步维艰。

这里所有关于“拥抱变革”的讨论都很美好，但如果每修改一行代码的代价是500美元的话，这些变革恐怕根本就不会发生。

当我还是一名年轻的软件开发者的时候，Bob Martin那篇关于面向对象设计原则的原创论文对我产生了很大的影响，他让我以一种全新的视角审视了自己的代码，并发现了其中的问题，在那之前，这些问题对我来说似乎从来都不是问题。

现在，你们也看到了如何才能写出既能提供当前价值，又不会阻碍未来价值的代码，期待你们也能亲自实践这些设计原则，并将其应用到自己的代码中。

就像学习骑自行车一样，单纯靠阅读是无法掌握软件设计方法的。为了让我们从这本书中的获益最大化，亲自实践是必不可少的。我们需要亲自分析自己的代码，查看其中是否存在 Bob 所强调的各种问题，然后在重构代码的实践中修复它们。如果你在重构方面是个新手，那么你将从本书收获双重的宝贵学习经验。

我们得学会将书中的这些设计原则以及整洁架构融入自己的开发过程中，这可以大大减少新代码给我们带来的麻烦。例如，如果我们现在正在进行一次测试驱动的开发（TDD），就可以在每一次测试之后做一些设计审查，并及时整理我们的设计（这比事后再修复这些不良设计要省时省力得多）。或者，在提交代码之前，我们也可以邀请同事一起审查代码。另外，我们也可以研究在构建软件的管道中引入一些代码的“质量把关”机制，以作为防止架构设计不够清晰分明的最后一道防线。（如果你还没有设置构建软件的管道，现在是否可以考虑设置一个了？）

这一切的重中之重就是要讨论架构的整洁性，我们要在自己的团队中讨论它，在各种开发者社区中讨论它。保证软件质量是我们每个人的责任，在区分架构的好坏标准上达成共识是一件非常重要的事。

我们必须意识到，大部分的软件开发者是没有太多架构意识的。就像 25 年前的我一样，是更有经验的开发者让我了解了架构。一旦我们一头扎进了整洁架构中，就会花时间围绕着它思考问题，并玩转它。

虽然开发者所在的技术环境一直在不断地发展，但本书所讨论的这些基本设计原则几乎不会发生变化。我一点都不怀疑在你们把 *Lean JSON Cloud NoSQL for Dummies* 当废纸卖掉很多年之后，这本书会还留在你们的书架上。我希望这本书会对你们有很大的帮助，就像 Bob 那篇原创论文对我的帮助一样。

愿你们真正的编程设计之旅从这里开始！

—— Jason Gorman

2017 年 1 月 26 日

附录 A

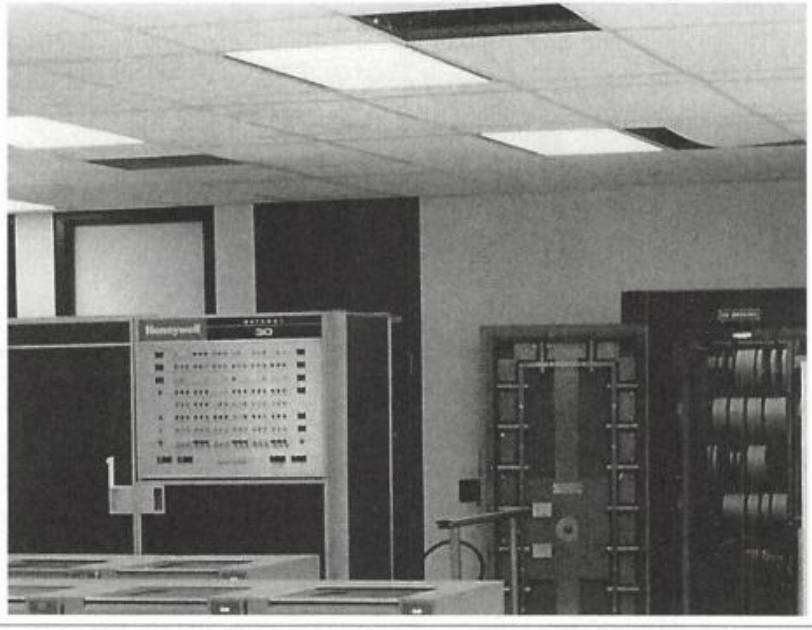
架构设计考古

为了说明这些优秀架构设计思想的来源，我打算和读者一起回顾一下我自己从1970年到现在所经历过的几个项目。其中，有些项目是在架构设计角度比较有特点的，而另一些项目则可以作为反例，指导后续项目。

另外，这个附录有一些个人自传体的影子，我会尽量将讨论重点集中在软件架构设计上，但难免会涉及一些其他方面的因素。

工会财务记账系统

20世纪60年代末期，有一家名为ASC Tabulating的公司与Teamster工会的705区签订了合同，要为其提供一套财务记账系统。在实现环境上，ASC公司选择的是一台GE Datanet 30计算机（参见图A.1）。



图A.1：GE Datanet 30

该图片来自Ed Thelen的网站ed-thelen.org。

我们从图 A.1 中可以看到，这台机器十分巨大[1]。它占据了整个房间，并且需要严格的环境控制设备。

这台计算机建构于集成电路发明之前，是用晶体管构建的。其中甚至还有一些真空管（虽然只是用在了磁带驱动器的感应放大器中）。

按照今天的标准来说，这台机器不仅体积大、速度慢、容量小，并且还非常原始。拥有 16K×18bit 的内核，7μs 的命令周期[2]。整台机器需要占据整个房间，还需要配备严格的环境控制系统。另外，这套系统还同时配备了 7 轨的磁带存储和一个能存储 20MB 的硬盘。

那块硬盘绝对可以称得上是一只怪兽。正如你在图 A.2 中看到的——可能仅看这张图还不能凸显它惊人的尺寸。基本上，其箱体已经超过我的身高，盘片直径 36 英寸，厚度 3/8 英寸。下面，我们可以通过图 A.3 来看看它的盘片。

数一下图 A.1 的照片中盘片的数量——该硬盘由十多个盘片组成，每个盘片有单独的寻址机器臂，由气动杆驱动。我们在其运转过程中可以观察到磁头寻址的过程，寻址时间在 0.5s 到 1s 之间。

这台机器开动时会发出飞机引擎一样的声音，在其运行稳定之前，地板都会随之摇晃震动[3]。

1　关于 ASC 的这台机器，我们听到的一个故事是，运送过程中它和一件家具同时被装载在一个拖车中。拖车在运送途中高速撞上了一座桥，整台计算机毫发无损，但是滑向前方把同车运送的家具挤成了碎片。

2　用今天的说法，时钟频率是 142 kHz。

3　请想象一下这块硬盘的整体质量，想象一下旋转它所需的动能！有一天我们进入操作室，发现柜下出现了一些金属碎屑。我们立刻给维护师打电话，他建议我们立刻关机。当维护师抵达现场之后，他说其中一个轴承已经磨损了，并同时给我们讲了一个故事：曾经有某块磁盘没有及时修理并马上替换轴承，导致盘片从系泊处脱离，击穿了一堵水泥墙，将停车场的一辆汽车劈成了两半。

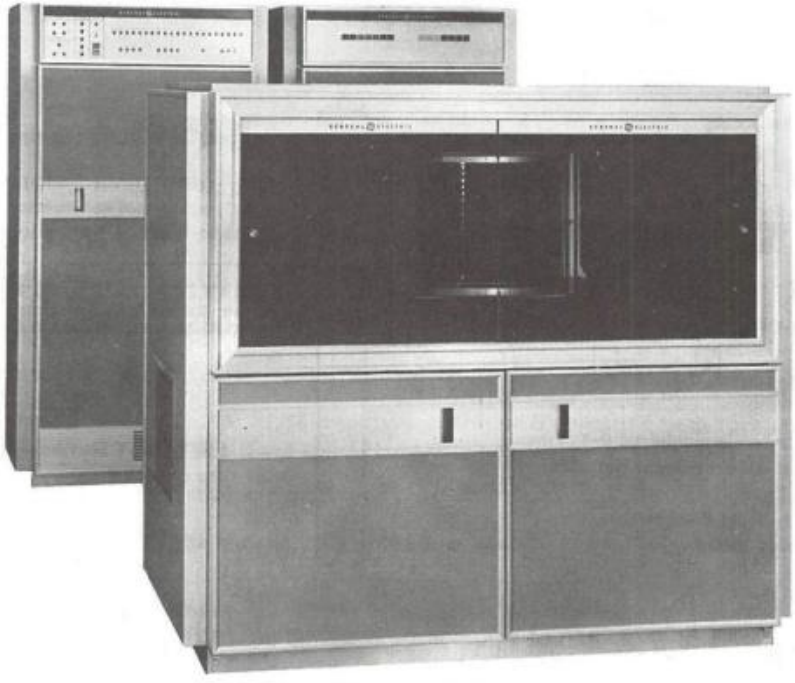


图 A.2：以盘片为单元的数据存储设备

图片来自 Ed Thelen 的网站 ed-thel0en.org。

Datanet 30 的一大卖点就是它可以相对高速地同时操作多个异步终端，这正是 ASC 所需要的。

ASC 总部位于伊利诺斯州的 Lake Bluff 市，距离芝加哥市三十英里，而 Local 705 办公室在芝加哥市中心。工会需要十几个数据输入专员使用 CRT[1]终端（图 A.4）来输入数据，另外还需要用 ASR35 电传设备（图 A.5）来打印报表。

1 全称为阴极射线管终端，是一种单色绿屏幕，只能显示 ASCII 字符的设备。

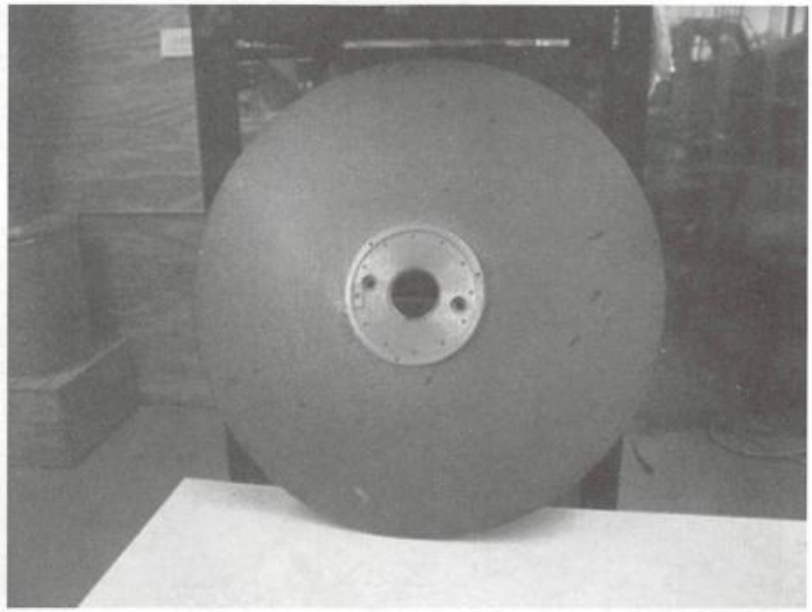


图 A.3：硬盘的盘片，3/8 英寸厚，直径 36 英寸

图片来自 Ed Thelen 的网站 ed-thelen.org。

当时的 CRT 终端每秒钟可以处理 30 个字符。这在 20 世纪 60 年代已经相当好了，因为当时的调制解调器都还没有达到这个速度。

为了满足该项目的需要，ASC 从电话公司租借了十几条专用电话线，以及两倍的 300 波特率调制解调器，用来将 Datanet 30 与这些终端连接起来。

这些计算机并没有预装任何操作系统，上面甚至都没有任何文件系统，只附带了一个汇编器。

如果我们需要在磁盘存储数据，就需要直接操作物理设备。那时候既没有文件，也没有目录的概念。程序需要计算出需要操作的磁道、盘片和扇区，同时要实际控制磁盘设备存取数据。是的，这意味着我们需要自己编写磁盘驱动程序。

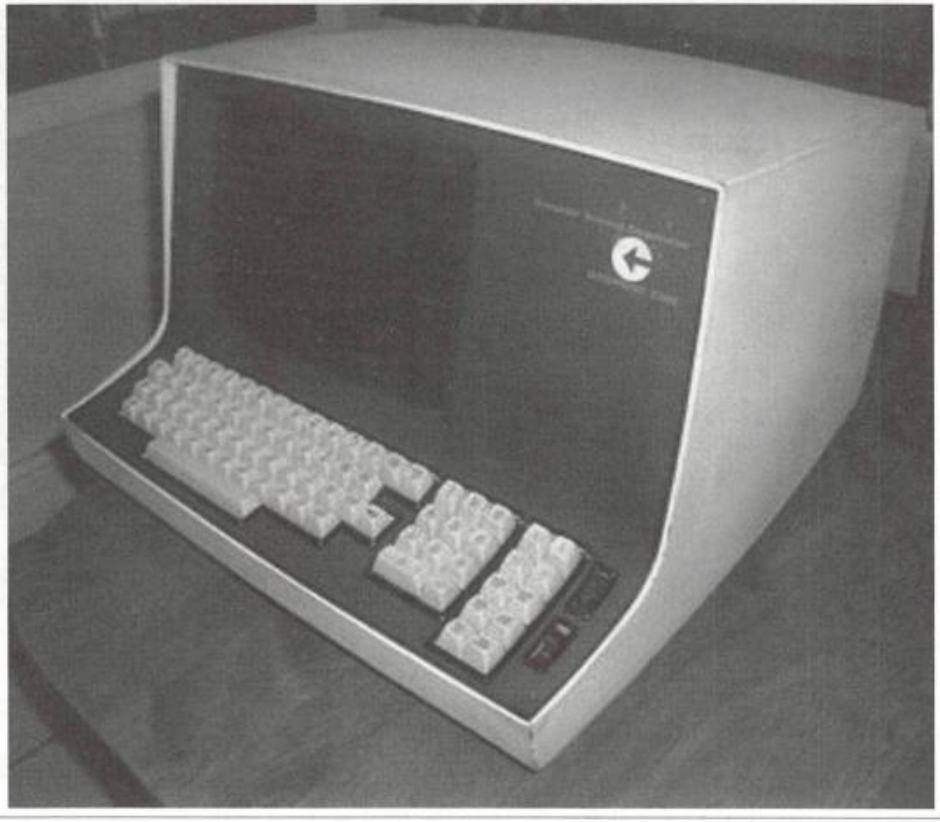


图 A.4：Datapoint CRT 终端

图片来自 Bill Degnan 的网站 vintagecomputer.net。

这个工会记账系统有三种数据记录格式：Agent、Employer 和 Member。该系统的基本功能是针对这些记录进行增删读改（CRUD）操作，但同时也包含付款提醒发送和记账等功能。

整套系统是一个咨询公司用汇编语言编写的，所有功能都挤在区区 16KB 的存储空间中。

不难想象，这个巨型的 Datanet 30 系统维护和运行成本非常高。负责维护软件的咨询公司当然也不便宜。而且，此时微型计算机逐渐流行，成本要低不少。

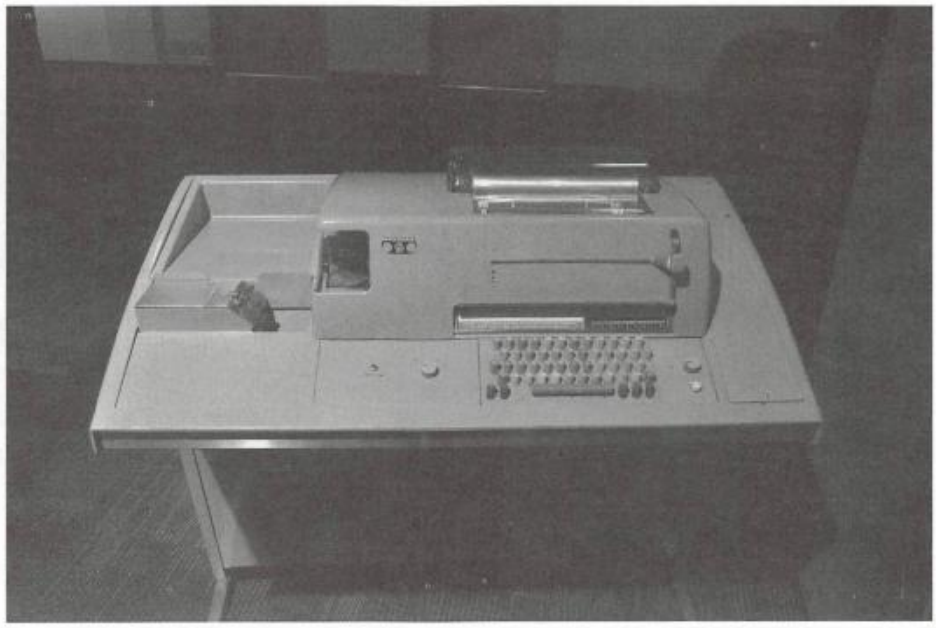


图 A.5：AR35 电传打印机

该图片由 Joe Mabel 提供。

在我 18 岁那年，即 1971 年，ASC 雇了我和我的两个极客朋友一起开发一套新的工会记账系统。这一回，我们使用的是 Varian 620/f 小型机（图 A.6），这种计算机造价很低，而且我们要的工资也不高，因此这对 ASC 来说很划算。

Varian 小型机具有 16 位的总线带宽，以及 32KB×16 的核心内存。该机器的计算周期大概是 1μs。整机性能大幅超越 Datanet 30，同时采用了 IBM 广受好评的 2314 磁盘技术，可以在 14 英寸直径的磁盘上存储大概 30MB 的数据。最关键的是，这种磁盘是穿不透水泥墙的！

当然了，该计算机上仍然没有操作系统和文件系统，更没有什么高阶编程语言。我们还是只能用汇编器凑合。

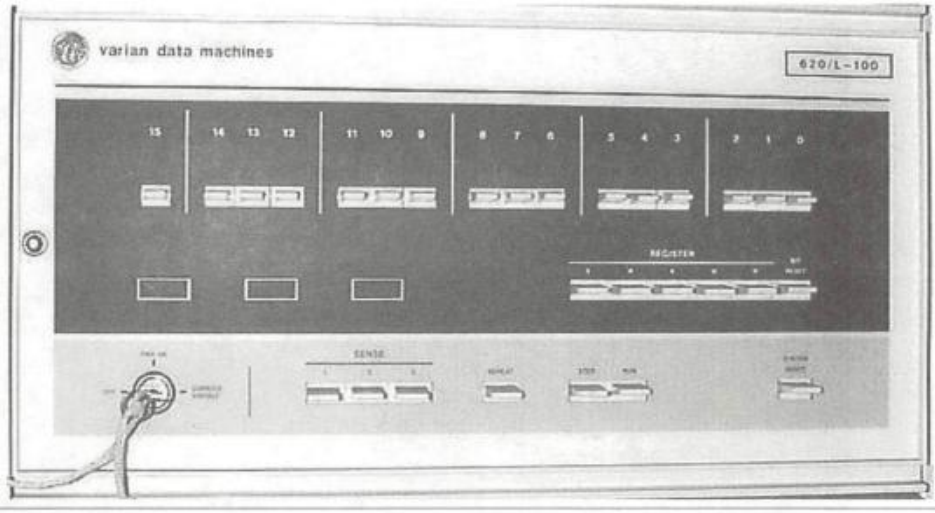


图 A.6：Varian 620/f 小型机

该图片来自 The Minicomputer Orphanage。

当时，我们没有采用将所有的系统挤在 32KB 内存空间中的做法，而是创造了一种交换空间技术。也就是说，我们让应用程序从磁盘加载到内存中来执行，执行完成之后再抢先与本地 RAM 交换并写回磁盘，以腾出内存让其他程序执行。

简而言之，这种空间交换技术就是将程序（由磁盘）交换到内存中的指定区域，待其执行一段时间，填满输出缓冲之后，程序就会被交换回磁盘，空出内存让另外一个程序执行。

当然，由于当时的用户界面的运行速度只有每秒 30 个字符，所以程序有充足的时间来执行换入/换出操作，从来没有人抱怨过响应时间的问题。

除此之外，我们还编写了一个抢占式任务管理器来统一处理中断和 IO。事实上，我们不仅需要编写应用程序，还需要编写磁盘驱动程序、终端驱动程序、磁带存储器的驱动程序以及整个系统中的其他一些程序。系统中的所有程序都是我们自己写的。那段时间我们经常每周加班工作 80 个小时左右，最终用了 8、9 个月的时间将这套系统搭建起来了。

整个系统的架构很简单（见图 A.7），即待一个应用程序运行到将其对应的终

端输出缓冲区填满时，抢占式任务管理器就会将其从内存中换出，同时将另外一个新的应用程序换入。同时，任务管理器会按照每秒 30 个字符的速度一点点将缓冲区中的内容输出给终端。当这些内容差不多输出完成的时候，再将应用程序换入，继续运行。

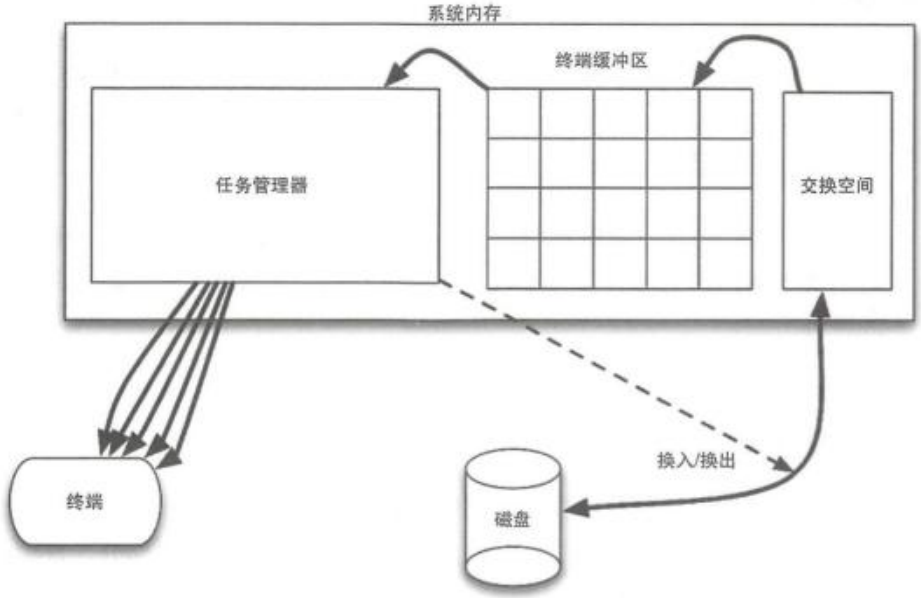


图 A.7：系统架构图

我们可以看到，该系统中存在两个架构边界。第一个架构边界是内容输出边界，应用程序并不知道其输出是发送给一个每秒显示 30 个字符的终端，输出部分对应用程序来说是完全抽象的。应用程序只要将字符串交给任务管理器，然后由任务管理器负责加载缓冲区，向终端发送内容，同时将应用程序换出/换入。

这个边界的依赖关系是正向的。也就是说，它的依赖关系与控制流在方向上是一致的。应用程序对任务管理器有编译期依赖，控制流也是从应用程序指向任务管理器的。边界阻止了应用程序知道输出到哪种设备。

第二个架构边界的依赖关系则是反向的。也就是说，任务控制器可以启动应用程序，但它对应用程序并没有编译期依赖，控制流是由任务管理器指向应用程序的。这个多态接口的实现方式其实很简单，它启动每个应用程序的方法都是直接跳转到

一个内存区域的指定位置。这个架构边界的作用是保证任务管理器只负责与应用程序约定一个启动内存位置，除此之外与应用程序无任何关联。

激光切割

1973 年，我加入了一家叫作 Teradyne Applied System（TAS）的公司，它是 Teradyne Inc.的一个子公司，总部在波士顿。这家公司当时的产品是一套公差精细的高能激光电子元器件切割系统。

在当时，电子设备制造商常会将丝状的电子元器件印在陶质的基板上，这些基板的大小通常在一平方英寸左右。这里说的电子元器件大部分是电阻——控制电流的设备。

电阻的性能取决于一系列因素，包括化学组成与几何形状。电阻的物理宽度越宽，阻碍效果越小。

我们所维护的系统需要操作陶质基板，以确保电阻元器件与两侧的探针接触。系统会持续测量电阻的性能，并用高能激光切去其不需要的部分，直到电阻性能与预期性能的误差小于 0.1%。

我们的主要生意就是向设备制造商出售这套系统，同时也直接承接一些小批量切削的工作。

该项目使用的计算机是 M365。在当时，很多公司都会自己组建计算机：Teradyne 自己构建了 M365 型计算机，提供给其所有的子公司使用。M365 是 PDP-8 的加强版——PDP-8 是当时非常流行的小型机。

M365 型计算机负责控制定位台，这个定位台主要用于标识陶质基板上探头所处的位置。M365 需要同时控制测量系统和激光系统，激光系统由一个计算机控制的二维反射镜系统定位，该系统同时还可以调节激光的功率。

M365 的研发环境很原始。整个系统没有存储系统，大容量存储主要依赖一些像老式 8 轨磁带那样的装置来完成，这些磁带和对应的驱动器是由 Tri-Data 制造的。

就像当时的 8 轨磁带那样，整个磁带是单向循环的。磁带驱动器只能单向转动磁带——没有回放一说！如果系统需要将磁带返回到开头，就必须不停向前快进，直至抵达“加载点（Load point）”。

磁带的卷动速度大概是每秒 1 英尺。因此，如果磁带总长 25 英尺，那么卷动到加载点最多需要 25 秒。为了处理这种情况，Tri-Data 制作了不同长度的专用磁带，从 10 英寸到 100 英寸的都有。

另外，M365 型计算机正前方有一个按钮，按下之后会将一段初始化程序加载到内存中。然后，这段初始化程序会读取磁带上的第一块数据并执行它。通常情况下，这段数据里会包含另一段程序，它会负责从磁带上加载整个操作系统。

操作系统随后会向用户提示输入需要运行的程序名称。这些程序也是存储在磁带上的。也就是说，加载完操作系统之后。用户需要输入他想要运行的程序名称——如 ED-402 Editor——操作系统随后会在整个磁带上搜索这段程序，然后将它加载并执行。

M365 型计算机的终端监视器是一个绿屏的 ASCII CRT 监视器，屏幕共有 24 行，每行 72 个字符宽[1]，该系统只能显示大写英文字母。

如果我们想要编辑一段程序，就需要先加载 ED-402 编辑器，同时将存有源代码的磁带插入。编辑器会将磁带上的第一块源代码数据读入内存并显示。每一块磁带数据中一般包含有 50 行源代码。编辑过程和使用 VI 很类似，需要操作游标并且进行修改。编辑完成之后，整块数据会被存储在另外一个磁带上，再从原磁带继续加载和编辑，直到所有代码都编辑完成。

在这个过程中，我们是不能回退到上一块的。因为当时的代码编辑过程必须是单向的，只能一次性从头写到尾。如果想要从头开始，则必须先将当前源代码完整拷贝到输出用的磁带上，再从头开始。不难想象，基于这种原因，我们在修改程序的时候一般都先在纸上用红色标记修改代码，之后，再一块块地将它们写入计算机。

在程序编辑好之后，我们需要返回到操作系统中并启动汇编器。汇编器读取源

1 “72”这个数字源自 Hollerith 打孔卡，这种卡片是 80 个字符宽。最后 8 个字符是保留给序列号用的。

代码磁带，将二进制代码输出到另外一个磁带上，同时使用数据打印机输出一份。

但由于磁带也并不是100%可靠的，所以当时我们总是会编译两次，这样可以保证至少有一个磁带是没有错误的。

当时的代码长度大概是 20 000 行，编译一次差不多需要 30 分钟。而且在这期间发生磁带读取错误的概率差不多是 1/10。如果汇编器读取磁带发生错误，会在终端上振铃提示，同时在打印机上输出一系列错误信息。这种让人抓狂的错误振铃提示在实验室中时常响起，当然，同时经常还会伴随着一个可怜的程序员的咒骂声。

当时的应用程序在架构上都很类似，它包括一个总执行程序（MOP），负责管理基本的 IO 功能；同时会提供一个比较原始的 Shell 界面。很多 Teradyne 的子机构都共享这一套 MOP 源代码，但每个机构随后根据自己的需要都对其进行了修改。于是我们会彼此共享修改过的版本，小心地将彼此的修改手工集成。

尽管当时确实存在着一个特定的工具层，专门用来控制测量硬件、定位台以及激光设备，但这些代码与 MOP 之间的边界只能说很模糊。虽然主要是工具层在调用 MOP，但 MOP 也为此做了很多特殊修改，并且常常回调工具层代码。是的，我们几乎不能把这两层代码当成独立的层。对我们来说，这个工具层只是 MOP 的一个高度耦合的延伸代码。

下一个要说的是隔离层。该层为应用程序提供了一个虚拟机接口，使得应用程序可以使用一个完全不同领域的编程语言 DSL 书写。该语言中包含了一些常用操作，例如移动激光头、移动定位板、切割等。客户使用这套语言来书写他们的激光切割程序，隔离层负责执行这些代码。

这种做法的真实目的其实并不是创造一个与机器无关的激光切割语言，而且这套语言其实和下层的某些部分紧密耦合。这里的真正目的就是让负责应用程序开发的程序员们使用比一种 M356 汇编语言更简单的方式去编写切割任务。

然后，这些切割任务将由操作系统负责加载并执行。实际上，我们开发的这个操作系统就是专门针对这个切割应用程序的。

整套系统是用 M365 汇编器语言来编写的，并且是作为一个单独的编译单元来

编译的，产生的是带有绝对地址的二进制代码。

这个应用程序的边界是相对较弱的，系统代码与用 DSL 编写的应用程序代码之间的关系也没有做明确的区割，耦合随处可见。

但对 20 世纪 70 年代的软件来说，这很常见。

铝压铸监控

20 世纪 70 年代中期，OPEC 组织正在推行石油禁运，石油短缺导致司机们在加油站争抢之中大打出手。我就是从那时候开始在 Outboard Marine Coporation（OMC）公司工作的，这是 Johnson Motors 和 Lawnboy lawnmowers 的母公司。

OMC 在伊利诺伊州的 Waukegan 有一个大型工厂，可以为所有公司的机械产品提供铝压铸部件。铝在大型熔炉中融化，装在大桶里运给数十个独立运转的铝压铸机器。每台压铸机由一个操作员来操作磨具，进行压铸操作，并且取出成品。这些操作员都是以其产出的部件数量来计算工资的。

我的工作是参与他们车间的自动化项目。OMC 公司为此购买了一台 IBM System/7——这是 IBM 用来应对小型机市场的产品。这台机器会与压铸机相连，以便我们统计机器的压铸操作次数和所需时间，最终展示在 3270 绿屏显示器上。

这里用到的编程语言仍然是汇编语言。同样的，所有执行的代码都是我们自己编写的。依然没有操作系统，没有子进程，也没有框架，只有最原始的代码。

同时，这套系统是中断驱动的实时系统。每次压铸机器完成一个操作周期，我们都需要更新一堆统计数据，同时给中央 IBM 370 发送消息，这套机器使用 CICS-COBOL 程序来将这些统计信息展示在绿色屏幕上。

我恨透了这份工作！噢，工作本身其实很有趣，只是工作环境的文化让人接受不了。说一个最简单的事，这份工作居然还会强制我天天戴领带！

我真的很努力，真的。但是我工作得实在太不爽了，我的同事们都知道。因为我从来记不住关键的交付日期，甚至忘记早起参加重要会议。这是我唯一一份被开

除的编程工作——当然，那确实也是活该。

从系统架构角度来看，这段经历并没有太多特别之处——除了一点。System/7有一个特别指令 *set Program Interrupt*（SPI）。这条指令允许程序触发处理器中断，以便处理其他低优先级中断。而现在，在 Java 语言中，我们用 `Thread.yield()` 来达到同样的目的。

4-TEL

1976 年 10 月，我从 OMC 离职，回到了 Teradyne 的另一个子机构——然后我在这里工作了 12 年。在这期间我参与研发的产品叫作 4-TEL。这个产品的作用是每天晚上测试一个电话服务区域内部的所有电话线，生成一个需要维修的线路列表。同时，电话线测试员可以利用这个系统进行详细的定点测试。

这套系统最初的设计和激光切割系统的架构类似，整个系统作为一个单体程序，用汇编语言编写，内部没有任何显著的架构边界。但是我加入的时候，正处于改变的边缘。

这套系统是由一个服务中心（SC）的测试员使用的。每个服务中心覆盖很多中央办公室（CO），每个中央办公室则处理接近 1 万条电话线服务。拨号与测量硬件必须存放在 CO 办公室中，于是 M365 计算机也和它们放置在一起。我们将这些计算机称为中央办公室线路测试器（COLTs）。另外有一台 M365 放在服务中心，我们称之为服务区计算机（SAC）。SAC 有数个调制解调器，可以用来与 COLTs 通信，速率是 300 baud（即每秒 30 个字符）。

最初，COLT 计算机要负责所有的操作，包括所有的终端通信、菜单、报告打印等。SAC 只是从 COLTs 获取信息，汇总之后在屏幕上显示。

这种设计的问题在于，每秒 30 个字符的速度实在是太慢了。测试员们不喜欢盯着屏幕等待数据，尤其是在他们仅仅需要几个关键数据的情况下。同时，M365 的核心内存非常昂贵，而程序尺寸也不小。

这里的解决方案是将拨号以及测量的软件部分，与分析结果、打印报告的软件

部分分开。后半部分被迁移到 SAC 上，前半部分则保留在 COLT 上。这样一来 COLT 可以减少体积和内存使用，同时响应速度也提高了，因为报告直接由 SAC 生成。

这套方案大获成功。屏幕更新速度非常快（一旦链接到了某个 COLT 上），同时 COLT 的内存占用大幅减少。

这里的架构边界非常清晰，同时高度解耦合。SAC 和 COLT 之间只需要交换非常少的数据包。这些数据包由一个简单的 DSL 写成，代表了一些简单命令，例如 DIAL XXX 或 MEASURE。

M365 是从磁带加载程序的。这些磁带驱动器非常昂贵，并且不太可靠——尤其是在 CO 这种工业环境下。同时 M365 和 COLT 其他的电子设备相比是非常昂贵的。所以我们最后选择推进采用 085 微处理器来替换 M365 的项目。

新计算机由一块带有 8085 处理器的电路板、一块带有 32KB 内存的电路板，以及三个 12KB 只读 ROM 的电路板组成。这些电路板和测量硬件装在同一个外壳中，这样就不再需要 M365 那样的大型外壳了。

ROM 电路板上面有 12 块 2708 EPROM（可擦除可编程只读内存芯片[1]）。图 A.8 是这个芯片的例子。我们通过将这些芯片插入一种叫作 PROM 烧录机的特殊设备进行编程。这些芯片可以通过高强度的紫外线来进行擦除[2]。

我和我朋友 CK 一起将 COLT 使用的 M365 汇编语言程序转换成了 8005 汇编语言。整个过程是纯手工进行的，消耗了大概六个月时间。最终的 8085 代码大概有 30KB。

研发环境中有 64KB 的 RAM，这样我们很快就能将编译好的二进制文件加载到内存中进行测试。

程序正常工作之后，我们再将其烧录入 EPROM。整个程序需要 30 个芯片，每个芯片都有标签写着与电路板的对应关系。

1 是的，我明白这里是自相矛盾的。

2 这种芯片有一个小塑料窗，可以看见内部的硅芯片，同时可以用紫外线来擦除资料。

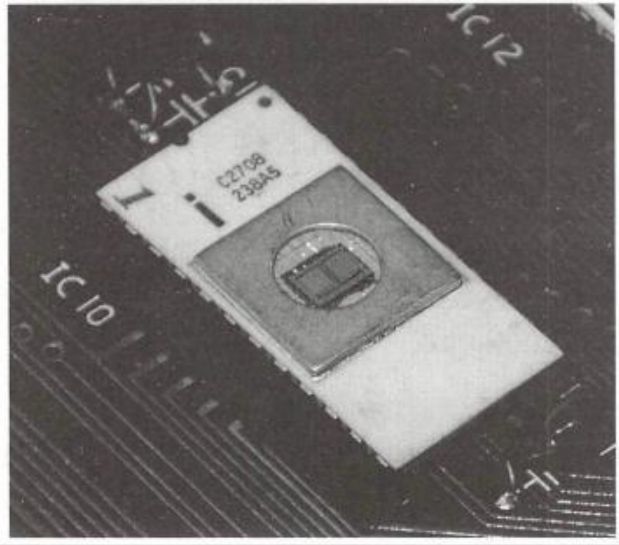


图 A.8：EPROM 芯片

30KB 大小的程序是一个单独的二进制文件，为了烧录到芯片上，我们简简单单将二进制文件划分为 30 个不同的段，每段大小 1KB，然后将每段烧录到正确的芯片上。

这套体系效果很好，我们很快开始大批量生产和部署这套系统了。

然而，软件总是“软”的[1]。总有新功能需要添加，总有 Bug 需要修复。随着部署的次数越来越多，每次更新都需要烧录 30 个芯片，还需要人工更换每个站点的设备，实在是太痛苦了。

这里面有各种各样的问题存在。有的时候芯片上的标签写错了，有的时候标签会掉落，有的时候外勤工程师会意外地换错芯片，有的时候外勤工程师会不小心把新的芯片上的针脚弄断。因此，工程师经常需要随身携带额外的 30 个芯片作为备份。

为什么我们每次都需要更换全部 30 个芯片呢？因为每次我们从 30KB 可执行文件里添加或者删除代码时，所有指令加载的地址都会发生变动。同时所有的函数与

1 是的，我知道当软件烧录入 ROM 的时候，就会被称为固件，然而固件也是很“软的”，因为经常需要改动。

子过程的地址也会改变。所以不管改动有多少，所有的芯片都要被替换。

有一天，我的老板过来找我，让我解决这个问题。他说我们需要一种方式，在不更换所有 30 块芯片的前提下修改代码。我们头脑风暴了一阵子，正式启动了“向量化”这个项目。这个项目花了三个月时间。

该项目背后的想法很简单。我们将这 30KB 的程序切分为 32 个独立编译的源代码，每个尺寸小于 1KB。在每个源代码文件的开始，向编译器声明编译结果所要加载的内存地址（例如，ORG C400 代表插在 C4 位置的芯片）。

同时，在每个源代码文件的最开始，将该芯片上所有的子程序的地址存放在一个简单的、固定大小的数据结构中。数据结构共有 40 个字节，能够存放 20 个地址，这也意味着每一个芯片可以存放超过 20 个子程序。

接下来，在内存中增加了一块特殊区域，称为向量区。该区域包含了 32 个表格，每个 40 字节——刚刚足够用来保存每个芯片开头的那个指针数据结构。

最终，我们将代码中所有的子函数调用都更改为通过内存进行的间接调用。

当处理器启动时，会扫描每个芯片，并且将每个芯片开始的向量表加载到内存中，最后跳转到主程序。

这样做的效果非常好。现在当我们修复一个 Bug，或者增加一个新功能的时候，我们只需要重新编译一个或者两个芯片的代码，现场维护工程师只需要替换这一两个芯片。

这样做实现了让每块芯片独立部署。我们发明了多态分发模式，发明了“对象”这个概念。

现在回头来看，这就是插件式架构，而且真的是物理可插拔的。最终，我们将这个概念推广，以至于可以通过在一个开放插座上插入一个芯片，就能实现新功能的安装。插入芯片之后，对应的菜单控制会自动出现，与主程序的绑定也会自动完成。

当然，当时还没有所谓的面向对象编程原则，我们也不知道区分用户界面与业

务逻辑的重要性。但是就是这么简单的一个修改，在当时也是非常有用的。

这个方法另外一个未曾预料到的好处是，我们现在可以实现拨号连接之后给固件打补丁。如果我们发现了固件中的一个 Bug，可以远程拨号连接到这台设备上，利用板上的监护程序修改内存中的向量表，将错误的子程序地址指向一块空余的内存。同时，直接将修复的子进程机器代码用十六进制数字直接写入内存。

这对我们的外勤工作来说太有用了，当然也对我们的客户来说很有用。如果遇到问题，他们不用预定紧急维护操作，也不需要我们马上运送新的芯片过去安装。系统可以线上打补丁，新的芯片可以在下次常规维护访问时安装。

维护中心计算机（SAC）

4-TEL 维护中心计算机（SAC）是基于 M365 迷你计算机开发的。这个系统与所有外地部署的 COLT 进行通信，通过专线或者拨号线路。该系统会命令这些 COLT 计算机测量每条电话线的状况，接收原始数据，进行一系列复杂分析从而识别和定位问题。

工单分派

这个系统工作的一个核心原则是保证修理工资源的合理利用。根据工会规则，修理工分为三个类别：中央办公室修理工、线路、客户修理工。中央办公室修理工只能维修中央办公室内部的线路问题，线路修理工负责维修中央办公室与客户之间的线缆问题，而客户修理工负责修理客户现场内部与线缆末端连接的问题。

当收到客户投诉时，我们这套系统可以远程分析问题，以决定派遣哪种类型的修理工。这样可以避免派错修理工而无法解决客户问题，造成浪费和延迟，更节省了电话服务公司大量的金钱。

负责决策派遣规则的这段代码是由一个非常聪明，但是非常不善沟通的人设计和实现的。传言这段代码的构建过程是这样的：“他花了三个星期盯着天花板构思，然后用两天时间拼尽全力写成了这段代码——接下来他就离职了。”

没有人理解这段代码。每次我们尝试加一段新功能，或者修复一个问题的时候，都会引入新的问题。由于这段代码事关整个系统存在的核心意义，每个新的问题都让公司上下蒙羞。

最终，管理层要求我们将这段代码封闭起来，不允许再次修改了。这段代码已经正式固化了。

这段经历让我从此以后对代码的整洁性深感重视。

系统架构

这个系统是在 1976 年用 M365 汇编语言编写的。作为一个单体程序，它差不多有 6 万行代码。操作系统是一个我们自己开发的、非抢占式的、依赖轮询的任务切换器。我们将其称为 MPS，即并行处理系统。M365 计算机没有预置堆栈，所以任务相关的变量都需要在一个特定内存区域中存储，每次上下文切换时会换入换出。共享变量用锁和信号量来管理。代码重入与竞争问题是很常见的。

当时，系统中没有对设备控制逻辑、UI 逻辑与业务逻辑代码之间的隔离。例如，调制解调器控制代码交织在业务逻辑与 UI 代码中间。没有任何模块化或者接口化的工作，调制解调器都是通过散落在各处的代码在比特层面直接控制的。

终端 UI 相关代码也类似。消息和格式化代码也没有隔离，散落在 6 万行代码中的各处。

我们当时使用的调制解调器硬件模块是设计为挂载在 PC 电路板上的。我们从第三方公司购买了这些模块，与其他的电路一起集成到我们自己设计的背板上。这些设备相当昂贵。几年过后，我们决定设计自己的调制解调器模块。我们软件组苦苦哀求硬件设计师，在设计新调制解调器时，一定要与老的组件采用同样的控制接口，一直到位级别。我们反复解释，控制代码散落在整个代码库的各处会导致难以修改，而且我们的系统未来还要同时处理两种不同的调制解调器。所以，我们反复不停地强调和哀求：“请一定要和原来的调制解调器采用同样的软件控制接口。”

然而，当我们实际拿到新的调制解调器时，控制结构是和以前完全不同的。不仅仅是有所区别，而是完全、彻底不一样。

谢谢啊，硬件工程师！

我们怎么办？这可不是仅仅将老的调制解调器替换为新的调制解调器，而是要混合部署到系统中。软件必须能够同时处理两种不同的调制解调器。我们是不是要在所有设备的代码中都增加各种特殊情况判断呢？这样的地方有几百个！

最终，我们采用了一个更糟糕的办法。

程序中有一个特定的子程序是用来向串口通信总线写入控制数据的，其中包括与调制解调器的通信。我们修改了这段代码，让它主动识别与老的调制解调器通信时候的字节位特征，同时将其转换为与新的调制解调器通信的代码。

过程相当复杂！向调制解调器发送命令的时候，需要向不同的 IO 地址发送不同的数据，我们的这段黑科技代码还要解析这些数据，然后按照原始顺序，将它们以完全不同的格式和不同的延时发往不同的 IO 地址。

最终这段代码还是正常工作了，但是这段程序真的是黑到不能再黑的黑科技。经历了这次事件之后，我深深懂得了将硬件代码与业务逻辑代码隔离——使用抽象层的重要意义。

大型重写

随着 20 世纪 80 年代的到来，生产自己的迷你计算机和设计自己的计算机架构逐渐失去了吸引力。市面上有很多迷你计算机可供选择，整合它们要比自己根据 20 世纪 60 年代的私有计算机体系架构来设计和实现更符合标准和省钱。另外，基于 SAC 软件糟糕的架构设计情况，我们的技术管理部门开始推动 SAC 系统的一次架构重新设计。

新系统计划采用存储在硬盘上的 UNIX 操作系统，用 C 语言编写，运行在 8086 微型计算机体系上。我们的硬件团队开始设计实现硬件平台，同时一小部分软件开发者组成了“虎之队”，负责重写软件部分。

我就不在这讲中间发生的许多故事了。简单来说，第一个“虎之队”在消耗掉 2 人年到 3 人年的成本之后，没有能够交付任何东西。

一年或者两年之后，可能在 1982 年左右，整个过程又启动了一次。这次的目标是重用 C 语言和 UNIX，在我们自己新设计的、非常强大的 80286 硬件上，将 SAC 整体重写。我们将这台计算机称为“深思者”。

这项工程耗费了几年时间，然后又延迟几年。我不知道第一套基于 UNIX 的系统是何时部署的，我认为直到我离开公司的 1988 年，也没有部署成功。可能这套系统最终也没有部署成功。

为什么项目总是延期？简单来说，对一个重新设计的团队来说，想要和一大群积极维护旧系统的团队保持一致是非常困难的。下面是他们遇到的其中一个困难的情况。

欧洲

在用 C 语言重新设计 SAC 项目启动的差不多同一时期，公司开始向欧洲进军。当然，不可能等待新的软件系统设计完成，所以很自然，基于老的 M365 的系统被用来在欧洲部署。

问题是，欧洲电话系统与美国电话系统非常不同，不仅如此，人员分工与管理结构也是完全不一样的。所以我们当中的一个最好的软件开发人员被派遣到了英国，在当地带领一个软件开发团队解决欧洲的问题。

当然，没有人认为能把这些修改集成到美国使用的版本中，别忘了，当时可没有可以跨大洋传递大量代码的网络基础。英国开发团队简单地将 US 代码复制了一份，根据需要自己修改了。

当然，这样造成了其他困难。大洋两侧的代码发现了同样的 Bug，需要在两地各自修复。但是由于模块在两地系统中已经被修改得面目全非，很难保证美国一侧的修改可以在英国系统中正常运行。

经过几年的折磨之后，同时也伴随着一条连接美国和英国办公室的高带宽网络线路的开通，我们开始了第一次整合两个分支的尝试，目标是将两套系统的功能区分变为配置文件的区分。这项工作第一次尝试失败了，第二次尝试又失败了，接下来是第三次尝试。两套代码，虽然十分相似，但是细节部分区别实在太多，无法简

单整合——尤其是在当时市场多变，两地都在不停修改的情况下。

同时，“虎之队”在尝试用 C 语言和 UNIX 来重写这套系统时，也意识到了这个重新设计需要同时处理欧洲和美国两地的情况，当然，这只会让其进度更缓慢。

SAC 结论

关于这套系统我还有很多故事可以讲，但是这实在是太让人郁闷了。简单来说，我软件研发生涯中的大部分教训都来自维护这套可怕的 SAC 汇编代码的经历。

C 语言

我们在 4-Tel 项目中使用的 8085 计算机硬件给了我们一个相对低成本的计算平台，我们利用它部署了很多工业环境的项目。该平台可以搭载 32KB 内存与 32KB ROM，同时有很丰富的配件控制能力。当时我们唯一缺少的就是一个灵活方便的编程语言，8085 汇编语言写起来真的很痛苦。

同时，我们当时采用的汇编器是自己开发的，运行在 M365 系统之上，还在使用前言“激光切割”一节中描述的磁带。

可能是命运使然吧！我们的首席硬件工程师说服我们的 CEO 购买了一台真正的计算机。他其实并不知道这台机器能用来做什么，但是他有很强的政治影响力。于是，我们购买了一台 PDP-11/60。

我，作为当时一个小小的软件开发人员，可乐坏了。我精确地知道我可以拿这台计算机来做什么，我认定了这台计算机要归我使用。

当使用说明先于机器几个月运到的时候，我将其带回家里如饥似渴地阅读。当计算机运到的时候，我已经非常了解如何在硬件和软件上操作了——水平至少在自学能达到的上限。

是我帮忙填写的购买订单，我在其中亲自书写了新的计算机所应该配置的磁盘

存储。我决定我们应该买两个磁盘驱动器，每个可以接受 25MB 的可替换磁盘。[1]

50MB！这个数字在当时看起来是近乎无限的！我还记得大晚上我在公司的办公室走道里像一个魔王一样狂笑："50MB，哈哈哈哈哈哈！"

我让行政部门专门构建了一个房间，以存放六台 VT100 终端。墙上贴满了太空图片，这就是软件工程师编写和编译代码的地方。

当机器运达时，我花了几天时间安装调试，给所有的终端接线调试——加班加点也乐意。

我们从 Boston Systems Office 购买了 8085 汇编器，将 4-Tel Micro 的代码转换成对应的格式。同时我构建了一套跨平台编译系统，使得我可以将编译好的二进制文件从 PDP-11 上下载到 8085 开发环境中，用来烧录 ROM。一切都和预想的一样。

C 语言

但是我们仍然还在使用8085汇编器。这可不是我想要的结果。我听说市场上有一种"新"的语言，大量运用于贝尔实验室，叫作"C"。于是我买了一本Kernighan和Ritchie写的*The C Programming Language*一书。就像几个月前阅读PDP-11手册那样，我把这本书翻烂了。

这个语言的优雅性将我彻底征服了。这门语言没有牺牲汇编语言的任何特性，但是同时又提供了一种更简明方便的使用方式。

我从 Whitesmiths 处购买了 C 编译器，在 PDP-11 上运行了起来。编译器的输出格式和BSO 8085编译器的输出格式兼容。于是我们实现了从C到8085硬件的贯通，一切太顺利了。

现在的最后一个问题就是如何说服一群嵌入式汇编程序员们使用 C 语言编程。这个噩梦一样的故事下次有机会再讲吧。

1　RKO7。

BOSS

我们使用的 8085 计算平台是没有自带操作系统的。根据我与 M365 系统中的 MPS 搏斗的经验，以及对 IBM System 7 平台中非常原始的中断机制的了解，我认为 8085 平台需要一个简单的任务切换系统。于是我设计了 BOSS：基础操作系统与任务调度器。[1]

BOSS 系统的绝大部分是用 C 语言编写的，提供了一种并行运行任务的能力。这些并行任务并不是抢占式的——任务切换并不是根据中断自动进行的。与之相反，和 M365 系统中的 MPS 系统类似，这些任务是基于简单轮询机制进行切换的。当某个任务阻塞，在等待某个事件时，轮询才会启动。

BOSS 中阻塞某个任务的调用如下：

```
block(eventCheckFunction);
```

该调用会将当前任务暂停，将 `eventCheckFunction` 加入轮询队列，将其与最新进入阻塞状态的任务对应起来。系统随后进行轮询，调用轮询列表中的每个函数，直到某个函数返回 `true`，该函数对应的任务就会恢复运行。

总的来说，正如我说的那样，这是一个简单的非抢占式的任务切换器。

该系统后来逐渐演变为许多项目的基石，下面我们来看其中的第一个项目 pCCU。

pCCU

20 世纪 70 年代末期到 80 年代初期的电信公司正处于动荡之中，其中一个大的变革就是数字化浪潮。

20 世纪以来，中央交换办公室与客户之间的线路一直就是一对铜双绞线。这些双绞线汇聚到一起组成了跨越全国的电话线网络，有时通过电线杆架在空中传输，

1 后来该系统改名为 Bob's Only Successful Software，Bob 唯一成功的软件项目。

有时埋在地下。

铜是一种珍稀金属，电话公司为了覆盖全国，需要储备数吨之多。这其中的资本投入是相当巨大的。而通过将电话服务转成数字信号传输，可以节省大部分的资本投入。单条铜双绞线可以以数字模式同时传输上百条电话通话。

基于这个趋势，电话公司开始了用现代数字化交换机替换传统模拟交换机的旅程。

我们的 4-Tel 产品主要测试铜双绞线链路，并不测试数字链路。在数字化环境中，也有很多线路需要测试，但是这些路线比以前要短得多，也更靠近用户。信号以数字模式从中央办公室传输到本地分发点，再转换为模拟信号模式，通过标准的铜线传递给用户。这意味着我们的测量设备现在需要迁移到铜线的位置，而我们的拨号设备还需要留在中央交换办公室。问题是我们之前设计的 COLT 是将拨号和测量功能集成到同一个设备中的。（如果我们能够早几年意识到这中间明显的架构边界，我们就可以省好多钱！）

因此，我们设计了一个新的产品架构：CCU/CMU（COLT 控制器和 COLT 测量器）。CCU 会放置在中央交换办公室，负责远程拨号功能。CMU 会部署在本地分发点，负责测量连接到用户的铜线。

问题是每个 CCU 都会对应很多 CMU，CMU 和电话号码的关系数据是存储在数字交换机之中的，CCU 必须和数字交换机交互才能联系 CMU 进行通信和控制。

我们承诺电信公司这个新架构一定能在过渡期间构建完成并正常运转，但是我们知道电信公司需要数个月，甚至数年才能完成这项改革，所以一点也不着急。同时，开发 CCU/CMU 硬件和软件也需要好几个人年呢！

排期中的陷阱

随着时间推进，我们发现总是有救不完的火，CCU/CMU 架构的工作只能一再推迟。但是我们并没有担心，因为电信公司也在不停推迟数字交换机的部署工作。根据电信公司的排期表，我们认为时间非常充裕，很有信心。

随后，某一天早上我老板把我叫到办公室："我们的一个客户下个月就要部署数字交换机了，我们必须随之交付一套正常工作的 CCU/CMU。"

我吓傻了！我们怎么可能在一个月之内完成需要几个人年的工作呢？但是我老板提出了一个想法……

事实上，我们并不需要一套完整的 CCU/CMU 架构。计划部署数字交换机的这家客户很小，只有一个中央办公室和两个本地分发点。更重要的是，所谓的本地分发点其实并不完全是本地化的。其中还是部署了传统的模拟交换机，用来给几百个用户提供服务。而且这些交换机正好是 COLT 可以呼叫的型号。更幸运的是，电话号码内部包含了足够的信息，可以用来判断需要联系哪个本地分发点。如果电话号码的某个位置是 5、6 或者 7，那么就联系分发点 1，否则联系分发点 2。

如同我的老板说的那样，我们其实并不需要一个 CCU/CMU。我们需要在中央办公室部署一个简单的计算机，通过调制解调器连接本地分发点的两个标准 COLT 设备。SAC 需要和中央办公室里新的计算机通信，该计算机负责解析电话号码，随后将拨号和测量命令分发到对应分发点的 COLT 设备上。

于是，pCCU 诞生了。

这是用 C 编写、使用 BOSS 操作系统的第一个实际部署的系统。整个系统花了我一周时间来开发。这个故事并没有什么系统架构设计的深意，但是对下一个项目来说却很重要。

DLU/DRU

20 世纪 80 年代初期，我们有一个得克萨斯州的电信公司客户。该客户需要覆盖的服务区域特别大，大到某一个服务区需要从几个不同的办公室派遣修理工。这些办公室都需要连接到 SAC 的终端。

你可能觉得这没什么大不了的——但是别忘了，这是 20 世纪 80 年代初，远程终端还没那么常见。更糟糕的是，SAC 硬件设计时假设所有的终端都是本地连接，靠一条私有的高速串行总线连接。

我们也有连接远程终端的能力，但是是基于调制解调器的，而且在 20 世纪 80 年代初，调制解调器的处理速度基本上不超过 300bps。我们客户对这么慢的速度一点也不满意。

高速调制解调器当时也存在，但是非常昂贵，而且必须运行在优化过的固定线路上。拨号连接的质量是远远达不到要求的。

我们必须提供一个解决方案，这个解决方案就是 DLU/DRU。

DLU/DUR 代表本地显示单元和远程显示单元。DLU 是一个插入 SAC 主板，伪装成一个标准终端控制器的设备。它不再使用串行总线，而是将字符流复用在一条单独的 9600 bps 的调制解调器链路上。

DRU 则是放置于客户位置的一个设备，它连接到 9600bps 链路的另外一端，然后将字符流解码再发送到对应的本地终端上。

看起来很奇怪吧？我们当时需要开发的东西，现在看来稀松平常到很多人可能都不会在意它的存在。但是这是很久以前了……

我们甚至还需要发明自己的传输协议，因为在当时，标准的通信协议并不是开源的。事实上，这比任何形式的因特网都要早。

系统架构

这套系统的架构非常简单，但是我想指出其中一些有趣的地方。首先，两个系统都采用了 8085 平台，都是用 C 语言和 BOSS 写成的。但是这并不是唯一相似的地方。

负责这个项目的共有两个人，包括我在内。我是项目负责人，Mike Carew 是我的帮工。我负责设计和编码 DLU，而 Mike 负责设计和编码 DRU。

DLU 系统架构是基于数据流模型的。每个任务负责处理自己的一小块业务，利用一个队列将输出传递给下一个任务，类似 UNIX 系统中的管道和过滤器模式。这个系统架构非常精巧，某个任务的输出可能会有多个任务同时接收，而多个任务的输出可能会汇总给一个任务处理。

这就像一条流水线。流水线上的每个工位都负责一个独立且专注的任务。产品从一个工位传递给下一个工位。有的时候流水线会分割成多条线路，有的时候多条线路会汇聚为一个单独的线路。这就是 DLU 的设计。

Mike 的 DRU 则采用了一种非常不同的设计。他为每个终端创建了一个任务，该任务负责这个终端的一切任务。没有队列，没有数据流，系统中仅仅包含许多等同的大任务，每个负责管理一个终端。

这与流水线模式正相反。可以将其比喻为，这个工厂聘用了很多技术专家，每个专家负责从头到尾打造自己的产品。

在当时，我认为我的设计是更好的，Mike 当然认为他的更好。我们经常对此辩论。最终，两个设计模式都工作得很好。这给我留下了深刻印象，软件架构可以看起来完全不同，但是仍然同样有效。

VRS

随着 20 世纪 80 年代的到来，更新的技术出现了。其中一个新技术是计算机合成语音。

4-Tel 系统的一个特性是帮助修理工定位线缆中的问题发生位置。流程大概如下：

- 中央办公室的测试员使用我们的系统来定位问题发生的大概位置，精确到英尺，系统误差在 20%左右。测试员随后派遣一名修理工到问题位置附近的修理点。
- 线缆修理员抵达位置之后，会呼叫测试员一同开始定位问题。测试员会启动系统中的故障定位程序，系统开始测量该链路中的电气指标，在屏幕上打印信息。系统会要求执行某些物理操作，例如断开或者短路该线缆。
- 测试员随后将系统需要的操作告知修理工，修理工完成操作之后告知测试员。测试员将信息输入系统，系统则进行下一步测量。
- 两到三个周期之后，系统就可以计算出一个新的问题定位点。线缆修理员则需要开车前往该地点重复上面的过程。

想象一下，如果线缆修理工可以在电线杆或者传输站独立完成这个过程，该有多好。这就是新的计算机语音控制系统发挥作用的时候了。线缆修理工可以直接拨入系统，用按键来发送命令，系统可以将信息用甜美的声音直接读给修理工听。

起名字

公司内部为这个系统的名字进行了一场小小的竞赛。一个最有创意的名字是“SAM CARP”，意思是“资本主义贪婪压制无产阶级的另一种表现形式”（Still Another Manifestation of Capitalist Avarice Repressing the Proletariat）。不用说，这个名字没有获选。

另外一个名字是“Teradyne 交互式测试系统”（Teradyne Interactive Test System）。这个也没有当选。

还有“服务区测试访问系统”（Service Area Test Access Network），也没有当选。

最后的获选者是“语音应答系统”（Voice Response System，VRS）。

系统架构

我并没有直接参与这个系统项目，但是我听说了其中的故事。下面我要讲的这个故事来自二手信息，但是我有充足理由认为它是真实的。

这是一个微型计算机、UNIX 系统，C 语言 SQL 数据库独领风骚的时代。我们信心满满，决定全部采用新技术。

经过对大量数据库的选择，我们最终选择了 UNIFY，这是一个可以在 UNIX 上运行的数据库系统，对我们来说正合适。

UNIFY 支持一种新技术，称为嵌入式 SQL。这种技术允许我们将 SQL 命令作为字符串嵌入 C 代码，于是我们就这么做了——而且嵌入得到处都是。

能在代码任意位置中嵌入 SQL 命令，听起来很酷，不是吗？于是我们就将 SQL 语句放到了代码的各处！

当然了，当时 SQL 还并不是一个稳定的标准。供应商总有很多特殊的怪癖。相应地，特殊的 SQL 和特殊的 UNIFY API 调用也就散布到了代码的各处。

系统完全正常！项目大获成功。修理工每天使用，电信公司很喜欢。一切看起来都那么美好。

随后，我们用的 UNFIY 产品突然宣布停止开发了。

哦？哦！

我们随后决定切换到 Sybase，或者 Ingress，我记不清了。但是不难想象，我们需要在 C 代码中不停搜索，找到所有嵌入的 SQL 和特殊的 API 调用，一个一个替换为新数据库需要的方式。

尝试三个月之后，我们放弃了。这根本不可能。我们的数据库与 UNIFY 结合得太紧密了，最终没有任何可能完成一次重构。

于是我们聘请了一个第三方公司专门维护 UNIFY，签署了一个维护合同。当然了，维护合同的价格每年水涨船高。

VRS 总结

这就是让我意识到数据库应该只是一个与业务逻辑隔离的实现细节的项目。这也是为什么我非常反感与第三方软件系统强耦合。

电子前台

1983 年，我们公司身处计算机系统、电信系统及语音系统的交集之中。CEO 认为这可能是研发新产品的好时机。为了达到这个目标，他组建了一个三人特殊团队（包括我），来构思、设计和实现一个新产品。

用了没多久，我们就想出了 ER（电子化前台）这个系统。

想法很简单，当呼叫某个公司时，ER 会首先接起电话，问你需要和谁通话。客

户用按键拼出对方的名称，ER 就会将电话转过去。ER 系统的用户可以从世界上任何地方拨入系统，利用简单的按键操作告知系统自己的电话号码。同时，系统也支持每个人存有多个不同的电话号码。

当呼叫 ER 系统并输入 RMART（我在系统中的代码）时，ER 就会呼叫我电话号码列表中的第一个。如果我没有回应，系统会呼叫列表中下一个号码，依此类推，直到最后一个。如果我还是没有接听，系统会提示呼叫者留言。

随后，ER 系统会定期呼叫我，尝试向我播放这条留言，以及这段时间内收到的其他留言。

这是世界上的第一个语音留言系统，我们[1]拥有专利。

整个系统的所有硬件都是我们自己构建的——计算机主板、内存、语音系统和电信系统，等等。整个系统的主电路板基于“深思者”——前文提到的 80286 处理器。

每块语音系统电路板支持一条电话线。包含一个电话接口、一个语音编码/解码器、一些内存和 80186 计算机。

计算机主板的软件系统是用 C 语言编写的。操作系统是 MP/M-86，一个早期的命令行驱动、多任务处理、基于磁盘的操作系统。MP/M 是 UNIX 的乞丐版。

语音系统的软件是用汇编语言编写的，没有使用操作系统。深思者与语音系统的通信是通过共享内存实现的。

整个系统的架构在今天看来，可以称为是*面向服务的*。每个电话线都由运行在 MP/M 下面的一个监听进程监控。

当一条呼叫进入系统时，系统启动一个起始处理进程，负责处理呼叫。随着呼叫从一个状态切换到另外一个状态，对应的处理进程会接管这条呼叫。

消息在服务之间以磁盘文件方式传递。当前运行的服务会判断下一个服务是哪

1　专利由公司持有。我们的雇佣合同里写得很清楚，我们发明的一切东西都归属公司。我老板告诉我：“你以 1 美元的价格将专利出售给了公司，而公司没有付这 1 美元给你。”

个，同时将必要的状态信息写入磁盘，使用命令行启动下一个服务，随后退出。

这是我第一次构建一个类似的系统。是的，这也是我第一次作为整个产品的系统架构师。我负责软件系统的一切事情——而且使之完美运转！

回头看看，我不能说当时的系统设计达到了本书的“整洁”标准，系统并不是一个插件式架构。然而，这个系统的确具有比较明显的架构边界。每个服务都是可独立部署的，仅仅关心自己领域内的事情。系统中有高阶进程和底层进程之分，而且依赖关系大部分是正确的。

ER 的消亡

不幸的是，这个产品的市场推广并没有取得良好成果。Teradyne 是一个销售测试设备的厂商，并不知道如何切入办公设备市场。

在不停地尝试两年之后，我们的 CEO 放弃了——不幸的是——同时放弃了专利申请。这项专利最终被市场中一个晚于我们三个月的公司拿到了。我们因此将整个语音留言和电话转移市场拱手让人。

唉！

不过，现在你们也不能再因为这些天天惹人烦的小机器而怪我了。

修理工派遣系统

ER 作为一个产品虽然失败了，但我们还是可以利用这个系统的硬件和软件来增强我们现有的产品线。更重要的是，VRS 系统的大获成功说服了我们应该提供一个与现有测试系统无关的专门针对修理工的系统。

于是 CDS——修理工派遣系统诞生了。CDS 其实就是 ER，但是 CDS 仅仅关注派遣电信修理工这个非常狭小的领域。

当电话线发生问题时，服务中心会创建一个问题工单。该工单由一个自动化系统管理。当修理工完成一个工作时，他可以呼叫服务中心来获取下一个工单。服务

中心操作员从系统中获取下一个问题工单，并且将它读给修理工。

我们的目标是让这个过程自动化。项目目标是让修理工可以呼叫 CDS 直接请求下一个任务。CDS 会与工单系统交互，读出结果。CDS 同时也会记录修理工与工单的关系，负责在工单系统更新维修状态。

这个系统中与工单系统交互、与厂房交互，以及与其他自动化测试系统的交互，是比较值得一提的。

在 ER 开发过程中采用的面向服务的系统架构经验使得我想更主动地推进这个设计。工单的状态机比 ER 中电话的状态机复杂得多。于是我开始了创造现在所谓的“微服务架构”的旅程。

每个电话呼叫的每次状态转换，不管多微小，都需要系统启动一个新服务。是的，状态机本身其实是用一个系统读取的外部文本文件维护的。系统中的呼叫发出的每个事件都会触发有限状态机的一个状态转换。现存进程会根据状态机状态来启动一个新的进程，随后要么继续等待某个队列，要么退出。

这个外部状态机文件允许我们在不修改任何代码的情况下，修改应用程序控制流（开闭原则）。我们可以很容易地添加一个新的服务，而不影响其他服务，用修改包含状态机的文本文件的方式将其嵌入系统控制流。我们甚至可以在系统运行的状态下进行这种操作。换句话说，我们实现了热替换和一套事实上的 BPEL（业务流程语言）。

旧的 ER 系统基于磁盘文件的服务通信方式，这对这种快速切换的服务来说太慢了，所以我发明了一个共享内存机制，称为 3DBB[1]。3DBB 允许以名字访问数据，这里的名字就是每个状态机实例所对应的名字。

3DBB 用来存储字符串和常量很方便，但是不能用来存储任何复杂的数据结构。这里面的原因是纯技术性的，但是很容易理解。每个 MP/M 的进程都运行在自己的内存分区中，一个内存分区的内存指针在另外一个分区中是没有意义的。由于此原

1　即 3D 黑板（Three-Dimensional Black Board），出生于 20 世纪 50 年代的人会知道这代表什么：Drizzle, Drazzle, Druzzle, Drome。

因，3DBB 中的数据不能包含指针，字符串没问题，但是树、链表或者其他带有指针的数据结构就不行了。

工单系统中的工单是来自很多不同的数据源的。有的是自动产生的，有的则是手工产生的。手工产生的工单是操作员与客户直接对话时手工创建的。根据用户的问题投诉，操作员会将他们的投诉转化为一系列结构化字符串，如下所示：

```
/pno 8475551212 /noise /dropped-calls
```

你应该已经看明白了，/字符意味着一个新话题的开始，/字符后面是代码，紧跟代码的是参数。代码成千上万，每个工单的描述中的代码段可能就有数十个。更糟糕的是，这些代码是手工输入的，所以有时候会出现拼写错误或者格式错误。这种代码只适合人类解读，并不是让机器处理的。

我们的一个困难之处是，如何解析这些半自由格式的字符串——需要解析并且修复任何错误，并且将其转化为语音播放给站在电线杆上戴一个耳机的修理工。所以这就需要我们的系统具有一个非常灵活的解析器，以及数据表达格式。这种格式必须通过 3DBB——一个只能处理字符串的系统传输。

于是在拜访各路客户途中的飞机上，我发明了一个被称为 FLD（外场标记数据）的格式。现在我们会称之为 XML 或者 JSON。具体格式当然是不一样的，但是背后的理念相通。FLD 是一个名字对应数据的带层级的二叉树。FLD 可以用简单的 API 快速查询，也可以和字符串互相转换，以便与 3DBB 对接。

你看，微服务通过共享内存（类似于 Socket）用 XML 等同的格式进行通信——这一切都是在 1985 年发生的。

太阳下面是没有新鲜事的。

Clear Communications 公司

1988 年，一批 Teradyne 员工离开了公司，创立了一个叫作 Clear Communications 的创业公司。几个月后我加入了他们。我们的目标是给一个监控 T1 线路通信质量的

系统创建软件——T1 线路是用来长距离传输数字信号的一种链路。公司的愿景是在一个大显示器上展示整个美国地图，其中每条出现问题的 T1 线路都会闪红灯。

别忘了，在 1988 年图形用户界面还是新事物。苹果电脑刚刚问世 5 年，Windows 还在萌芽期。但是 Sun 公司的 Sparcstations 具有可用的 X-Window GUI，所以我们最后采用了 Sun 架构——包括 C 语言和 UNIX。

这是一个创业公司，我们每周工作 70 到 80 小时。我们有愿景，也有足够的动力、意愿和能量。我们有专业知识，有公司的股权，怀抱着成为百万富翁的梦想每天努力工作。

我们产出了大量的 C 代码，修建了一个精美的空中楼阁。我们采用了进程、消息队列，以及超高规格的大型系统架构设计。我们甚至完整实现了 ISO 7 层通信协议栈——一直到链路层。

我们写了很多 GUI 代码。同时我们也写了很多烂泥巴代码，很烂很烂的泥巴代码。

我个人就亲手写过一个 3000 行 C 代码组成的 `gi()` 函数。从名字就能看出这是图形化解释器。这是烂泥巴代码的神作。这不是我写的唯一的烂泥巴代码，但是可能是最臭名昭著的。

还谈什么系统架构？开玩笑吧，这是一家创业公司！我们没有设计系统架构的时间，赶紧写代码吧！为了生存！

于是我们写了又写，然而三年过后我们却没能做好销售。我们有那么一两个客户，但是整体来说市场并不同意我们的远大愿景，我们的风险投资人也失去了耐心。

这是我人生的最低谷，我所有的努力和梦想都破碎了。我不仅工作不顺利，还由于工作导致家庭不顺利，整个人生都混乱了。

这就是我接到那个改变我余生的电话的时候。

蓄势待发

在这通电话发生的两年前，有两件大事发生。

首先，我构建了一套系统，用 UUCP 连接到附近的一家公司，该公司连接到另外一家公司，最终连接到互联网。这些都是拨号连接。当然了，我们的主 Sparcstation 工作站（就是我桌上的这台）使用一个 1200bps 的调制解调器，每天连接两次 UUCP 主机。这样我们可以接收 E-mail 和 Netnews（早期的一种社交网络）。

其次，Sun 发布了一个 C++编译器。从 1983 年起，我就对 C++和面向对象编程非常感兴趣，但是当时编译器很难找。当这个机会出现时，我第一时间就换了语言。我抛弃了 3000 行的 C 函数，开始编写 C++代码，这就是问题的开始……

我读了大量的书籍，当然，我读了 Bjarne Stroustrup 写的 *The C++ Programming Language* 和 *The Annotated C++ Reference Manual*，我读了 Rebeccas Wirfs-Brock 写的关于责任驱动设计的好书：*Designing Object Oriented Software*，我还读了 Peter Coad 写的 *OOA*、*OOD*、*OOP*，我还读过 Adele Goldberg 写的 *Smalltalk-80*，我还读过 James O.Coplien 写的 *Advanced C++ Programming Styles and Idioms*。也许最重要的是，我读了 Grady Booch 写的 *Object Oriented Design with Applications*。

多么美妙的名字！Grady Booch。谁能忘记这么美妙的名字呢？更重要的是，他在一个名叫 Rational 的公司担任 Chief Scientist！我也想成为首席科学家！我反复阅读他的书，学啊学啊！

随着我学习的深入，我开始在 Netnews 上参与辩论，和今天大家在 Facebook 上掐架一样。我讨论的主题集中在 C++和 OO 上。整整两年时间，我通过和 UseNet 上的几百人讨论最佳语言特性和最佳设计原则来缓解工作中的不爽。一段时间之后，我居然开始能够自圆其说了。

在某个讨论中，SOLID 原则的一些基础逐渐成形了。

所有的这些讨论，也许只是其中的一些信息，引起了某些人的注意……

Bob 大叔

Clear 公司里有一个年轻工程师，名叫 Billy Vogel。他给所有人都起了外号，我的外号就是 Bob 大叔（Uncle Bob）。虽然我的名字叫 Bob，但是我怀疑他是在即兴引用 J.R. "Bob" Dobbs （参见 https://en.wikipedia.org/wiki/File:Bobdobbs.png）。

一开始我没当回事，但是随着时间的推移，在他讨人厌的“Bob 大叔……Bob 大叔”的压力下，加上对公司的日益失望，让我萌生了退意。

接着，有一天，电话铃响了。

一通电话

拨来电话的是一个招聘专员。他联系我的原因是知道我是一个懂得 C++和面向对象设计的人。我不知道他具体是如何知道这个信息的，但是我怀疑很有可能和我在 Netnews 上的辩论有关。

他说有一个在硅谷的工作机会，一个名叫 Rational 的公司，正在寻找一起构建 CASE[1]工具的人。

我脸都白了，不用说我都知道发生了什么，这是 Grady Booch 的公司！我得到了一个可以和 Grady Booch 共事的机会！

ROSE

1990 年，我以合同工的方式加入了 Rational，参与 ROSE 产品的开发。这是一个让程序员画 Booch 图的工具——Grady 在 *Object-Oriented Analysis and Design with Applications* 一书中描述了这种图（图 A.9 是一个例子）。

Booch 表达法功能非常强大，直接为后续的 UML 等表达法做了铺垫。

ROSE 是具有架构设计的——而且是真正意义上的。该系统由多层组成，每层

1 全称为 Computer Aided Software Engineering，即计算机辅助软件工程。

之间的依赖关系是良好控制的。整个架构可以保证每层可以单独发布、研发，以及独立部署。

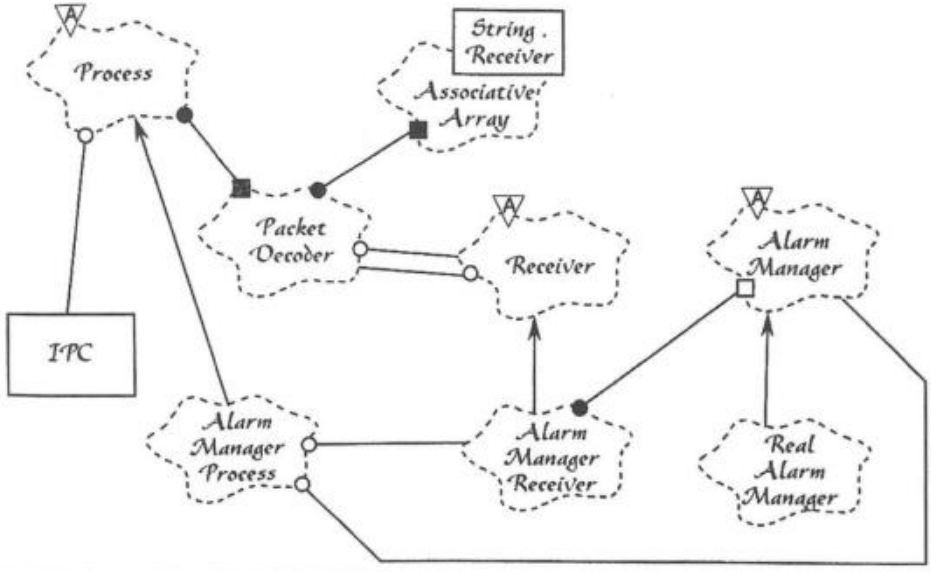


图 A.9：Booch 图

当然，它并不是完美的。当时架构设计中的很多原则我们并不完全掌握，例如还没有能够实现一个真正的插件式架构。

我们同时也不可避免地受了当时的一个潮流的影响——采用了一个所谓的面向对象的数据库。

然而，总的来说，这段经历是非常好的。我与 Rational 团队在 ROSE 项目上合作了一年半的时间。这段经历让我很受启发，是我职业生涯中最具代表性的一个阶段。

辩论仍在继续

当然，我也没有停止在 Netnews 上的辩论。事实上，我大幅度增加了上网的时间。我开始为 *C++ Report* 写文章。同时，在 Grady 的帮助下，我开始写第一本书：*Designing Object-Oriented C++ Applications Using the Booch Method*。

有一件事让我很不爽，我知道这么想不太正常，但我就是很不爽。再也没有人叫我“Bob 大叔”了。我发现我开始怀念这个名字。所以我不能免俗地把“Uncle Bob”放在了我的 E-mail 和 Netnews 签名中。这个名字就这么流传了下来。回过头来看，其实这是一个挺好的品牌名称。

……以另一个名义

ROSE 是一个巨大的 C++程序，分层开发，具有严格要求的依赖规则。这些规则和我在本书中描述的并不一样。我们没有将依赖指向高层策略，而是按传统模式指向了控制流的方向。GUI 依赖表现层，表现层依赖修改规则，修改规则依赖数据库。最终，这种错误的依赖指向方式很大程度上导致了整个产品的最终失败。

ROSE 的系统架构设计与一个优秀的编译器架构类似。图形表达式被解析成一个内部表现式（IR）。该表达式被一系列规则修改，最终存储于一个面向对象的数据库中。

面向对象的数据库是一个相对新的事物，整个 OO 世界都在不停讨论这其中的深远影响。每个面向对象程序员都想在自己的系统中使用面向对象数据库。这个想法虽然简单，但是却过于理想化了。数据库存储的是对象，而非表格。数据库和 RAM 类似。当访问某个对象时，它就神奇地出现在内存中，如果某个对象引用了另外一个对象，那么当你访问时另外一个对象就出现在内存中。一切都像魔法一样。

回头看，数据库部分可能是我们犯的一个大错误。我们本想要获得魔法一样的能力，但是最终我们实际得到的是一个庞大、缓慢、侵入性特别强、成本特别高的第三方框架，这个框架在各个级别上阻碍我们的进展，让我们的每一天都非常痛苦。

同时数据库也不是我们唯一的错误。最大的错误，事实上是过度设计。系统中存在许多许多的层，每个层都包含有自己的通信开销，这最终导致了整个团队的生产力大幅下降。

于是，在耗费了很多人年的工作量，经过无穷的挣扎，以及两个难产的发布之后，整个工具被抛弃了，被一个威斯康星州的小团队写的小程序替代了。

就是从这里，我领悟到大型架构设计有时候会导致大型灾难。作为系统架构师

必须要灵活地适应所遇到问题的规模。如果你只需要一个小的桌面应用工具，却采用大型企业式设计，肯定会导致灾难发生。

系统架构师注册考试

1990 年早期，我成为了一名真正的咨询师。我走遍了世界各地，给大家传授“面向对象”这个东西。我的咨询工作大部分集中在设计和架构面向对象系统上。

我的一个早期咨询客户是 ETS（教育测试服务机构）。该机构与国家架构师注册委员会（NCARB）签署了合同，负责考核新建筑架构师。

任何一个想要成为美国和加拿大注册建筑架构师（设计建筑物的架构师）的人都必须通过这项测试。该测试要求架构师应考者回答一系列与建筑设计有关的问题。该应考者需要根据一系列要求设计公共图书馆、餐馆、教堂，并且画出对应的架构设计图。

所有答题结果会收集起来，由一批资深设计师聚在一起作为评委，给答案打分。这个评分工作是一个耗资昂贵的大型活动，也是整个过程中出现延迟和意外的主要缘由。

NCARB 想要自动化整个过程，应考者在计算机上答题后，通过另外一个计算机来自动评分。NCARB 要求 ETS 研发整个软件系统，ETS 则聘用了我来带领一个团队完成这个工作。

ETS 将问题分解为 18 个测试章节。每个章节需要应考者用一个类似 CAD 一样的 GUI 程序描述自己的解决方案。另外一个独立的评分系统则根据这个结果评分。

我的同伴 Jim Newkirk 和我意识到 36 个应用程序具有极强的相似性。18 个 GUI 应用程序具有类似的操作方式和机制。18 个评分应用程序都使用相同的数学算法。基于这种相似性，我们决定给 36 个应用程序研发一个可重用的框架。基于这个思想，我们说服了 ETS——第一个应用会花很长时间制作，但是其他的应用只需要几周时间就可以搞定。

看到这里，你可能已经在掩面长叹，或者以头撞墙了。有类似经历的人都会记得当年 OO 的复用承诺是多么不靠谱。当时，我们可是深信不疑的，认为只要书写干净整洁的面向对象 C++代码，就可以产出很多很多的可复用代码。

于是，我们开始写第一个应用程序——所有程序里面最复杂的一个，被称为 Vignette Grande。

我们两个全部时间都扑在 Vignette Grande 上，时刻留心创造一个可复用的框架。这个工作花了一年时间。最终，我们产出了一个 45 000 行代码的框架和 6 000 行代码的应用程序。我们将其交付给了 ETS，ETS 则和我们签署了一个快速开发剩余 17 个应用程序的合约。

于是我和 Jim 招聘了三个程序员，组队一起开始制作其余的应用程序。

然而，意料之外的事情发生了。我们发现之前创造的所谓可复用的框架其实并不是完全能够复用的——在新应用程序中无法很好地匹配，总有微小的冲突引发问题。

这太令人失望了，但是我们自信知道如何解决，就跑到 ETS 和他们说工期需要延长——我们的 45 000 行代码的框架需要修改，至少需要小幅修改。这当然要花一定的时间。

不用说，你也知道 ETS 对此感受如何。

于是我们又开始了下面这段旅程。把旧的框架抛在一旁，开始同时写 4 个应用程序。我们从老的框架中截取一些可以重用的组件，以保证它们可以在 4 个应用程序中同时使用。这个工作花费了另外一年的时间，产生了另外一个 45 000 行代码的框架，以及另外 4 个各有 3 000~6 000 行代码的应用程序。

不用说，GUI 应用和框架之间的关系是符合依赖指向规则的。应用程序是框架的插件。高层的 GUI 策略都在框架中，应用部分代码只是胶水。

评分应用和框架的关系则相对复杂。高层的评分策略是存在于具体应用中的，而评分框架则是作为一个插件集成到评分应用中的。

当然，两个应用都是静态链接的 C++应用程序。所以所谓的插件概念并没有出现在我们的设计中。然而，这里面的依赖关系的确符合依赖指向规则。

交付了这 4 个应用程序之后，我们开始了接下来的 4 个。这次的确每个只花了几个星期时间，正如我们期待的那样。这段延迟几乎消耗了我们一年时间，所以我们额外聘请了另一位程序员来加速整个过程。

我们最终按时交付了。客户很满意，我们也很满意，一切都那么美好。

但是我们还是上了重要的一课：在真正制作出来一个可复用框架之前，是不知道怎么制作一个可复用框架的。想要制作一个可复用的框架，必须要和几个复用该框架的应用一起开发。

小结

如同我在本附录最初所说的，这个附录有一点自传的味道。我描绘了一些我认为对系统架构设计发挥了重要作用的项目，也描述了一些对我个人比较重要，但是对本书技术内容没那么重要的项目。

当然，这里的历史只是一部分。过去几十年中我还负责了许多许多其他的项目，我同时故意将历史停留在 20 世纪 90 年代早期——因为我接下来还有一本书专门讲述 20 世纪 90 年代末期的故事。

我希望读者喜欢这些故事，并且能从这些故事中获益。

Broadview
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

架构整洁之道

善用软件架构的通用法则，即可显著提升开发者在所有软件系统全生命周期内的生产力。如今，传奇软件匠师Robert C. Martin（Bob大叔），携畅销书*Clean Code*与*The Clean Coder*所获巨大成功之威，向我们深刻揭示了这些法则并亲授运用之道。

Martin在《架构整洁之道》中远不只是在为我们提供选项，他几乎是在将软件世界中横跨半个世纪的各种架构类型的经验倾囊相授，目的是让读者既能阅尽所有架构选型，又可通晓其如何决定成败。Martin也的确不负厚望，本书中充满了直接而有效的解决方案，以供读者应对自己面临的真正挑战——那些或最终成就或彻底破坏项目的挑战。

《架构整洁之道》不可不读，无论读者是现任的还是将来的软件架构师、系统分析师、系统设计师或软件项目经理，或是身负将他人设计落地重任的开发人员，这本书都可以让你们受益匪浅。

·本书内容·

- 了解软件架构师能力指标，以及达成这些指标所需的核心准则、实践。
- 掌握用于函数处理、组件分离与数据管理的必要软件设计原则。
- 了解编程范式如何通过限制开发者行为的方式强施纪律。
- 领会“至关重要”与“细枝末节”的区别。
- 实现针对Web、数据库、胖客户端、控制台与嵌入式应用的优选高级架构。
- 合理定义范围与层级，并对组件与服务进行组织。
- 理解设计和架构失败的原因，以及如何预防或修复这些失败。

·作者简介·

Robert C. Martin（Bob大叔），一位从1970年开始从业的老牌程序员，也是时常出席世界性会议的著名演说家，著有*The Clean Coder*、*Clean Code*、*Agile Software Development*及*UML for Java Programmers*等书。另外，Martin还是Uncle Bob Consulting LLC的创始人，以及Clean Coders LLC的联合创始人（与其子Micah Martin一道）。除此之外，他还曾任*C++ Report*主编、“敏捷联盟”首任主席，以及Object Mentor公司联合创始人及领导人。

Pearson
www.pearson.com

策划编辑：张春雨
责任编辑：付 睿
封面设计：侯士卿

上架建议：软件开发 / 架构

ISBN 978-7-121-34796-2

定价：99.00元