Broadview
www.broadview.com.cn

Pearson

架构整洁之道

Clean Architecture

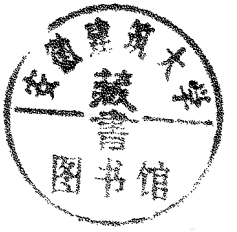
[美] Robert C. Martin 著
孙宇聪 译

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 9 月第 1 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翎的内容）。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（angstrøm，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。