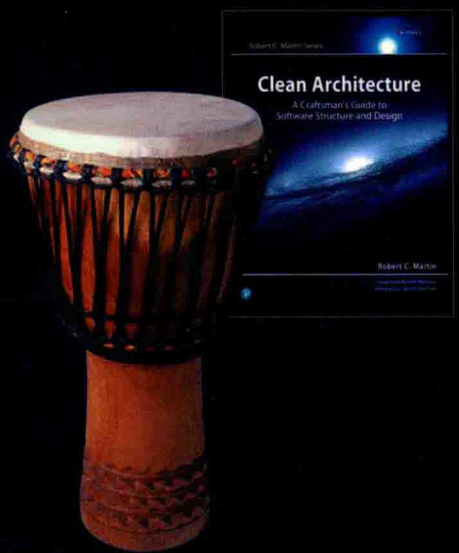
Broadview
www.broadview.com.cn
Pearson
· 原味精品书系 ·

# Clean Architecture（英文版）

## 软件架构与设计匠艺

Clean Architecture
A Craftsman's Guide to Software Structure and Design

[美] Robert C. Martin 著




感谢James Grenning与Simon Brown对本书的贡献
由Kevlin Henney作前序

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

· 原味精品书系 ·

# Clean Architecture（英文版）

## 软件架构与设计匠艺

Clean Architecture

A Craftsman's Guide to Software Structure and Design

[美] Robert C. Martin 著

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

## 内 容 简 介

通过合理运用软件架构的通用法则，可以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，深刻揭示这些法则并亲授运用之道。Martin 在本书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设计经验倾囊相授，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。Bob 大叔也的确不负厚望，本书中充满了直接而有效的解决方案，以供读者应对所面临的真正挑战——那些或最终成就或彻底破坏你项目的挑战。



版权贸易合同登记号　图字：01-2017-7529

**图书在版编目（CIP）数据**

Clean Architecture：软件架构与设计匠艺=Clean Architecture: A Craftsman's Guide toSoftware Structure and Design：英文 /（美）罗伯特·C·马丁（Robert C. Martin）著. —北京：电子工业出版社，2018.7

（原味精品书系）

ISBN 978-7-121-34261-5

Ⅰ. ①C… Ⅱ. ①罗… Ⅲ. ①软件设计－英文 Ⅳ.①TP311.5

中国版本图书馆 CIP 数据核字(2018)第 109377 号

责任编辑：张春雨

印　　刷：三河市华成印务有限公司

装　　订：三河市华成印务有限公司

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：26.5　字数：513 千字

版　　次：2018 年 7 月第 1 版

印　　次：2018 年 7 月第 1 次印刷

定　　价：109.00 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。

谨以此书献给我亲爱的妻子、可爱的儿女，

以及我的五个孙子孙女——我的心肝宝贝小甜心们！

# 前言

软件架构（Architecture）究竟指的是什么呢？

正向比喻是一种修辞手法，试图用架构的语言来描述某个软件，结果可粗可细，可能会过度描述，也可能会表达不足。

用架构来描述软件的明显优势是可以清晰地描绘其内在的组织结构（structure）。不管是在讨论组件、类、函数、模块（module）、还是层级、服务、微观与宏观的软件开发过程，组织结构都是一个主要关注点。但是真实世界中的许多软件项目并不是按我们的信念和理解生长的——它们底层层层嵌套，顶层则往往是一团乱麻，相互纠缠。有的时候真的很难让人相信，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，不管地基是石头还是水泥，高大还是宽阔，宏伟还是渺小，其组织结构都一目了然。物理建筑的组织结构必须被“重力”这个自然规律以及建筑材料自身的物理特性所约束。用砖头、水泥、木头、钢铁或者玻璃造就的物理建筑与软件项目相比，最大的不同点就是，大型软件项目由软件组件构成，而这些软件组件又由更小的软件组件构成，层层嵌套。

当我们讨论软件架构时，尤其要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（detail）设计，架构问题就无从谈起。大型物理建筑的组织架构常常是由其中一个个细节设计共同决定的，如果细节设计太多，那么组织架构就会更复杂，反之亦然。但是软件项目的复杂程度却不一定能用物理尺度来衡量。软件项目也有组织结构，不论从数量上还是种类多样性上都远远超过物理建筑。我们可以很明确地说，软件开发比修建物理建筑需要更多、更专注的设计过程，软件架构师比建筑架构师更懂架构！

虽然人类已经习惯了使用物理尺度来衡量和理解现实世界，但这些却不适用于软件项目。不管某个 PowerPoint 图表中的彩色方块多么好看、多么简单易懂，也无法完全代表一个软件的架构。它只能是某个软件架构的某种展现形式。软件架构并没有一个固定的展现形式，每一种展现形式都建立在背后的层层抉择之上，例如，哪些部分要包含其中，哪些则应该被排除；哪些部分用特殊形状和颜色进行强调，哪些部分则一笔带过，甚至直接忽略。每种表现形式都是对的，它们往往没有任何内在的联系。

虽然软件可能是人凭空编造出来的，但还是要在现实世界中运行。虽然在设计软件架构的过程中物理定律和物理尺度可能并不是主要考虑的对象，但我们还是要理解和遵循某些约束条件。CPU 速度和网络带宽往往直接决定了系统的性能。内存和存储的大小限制也会影响代码的设计。

> 这就是爱的不幸，期待是无穷的，但行动却是有限的，欲望是无止境的，体验却如奴隶般受限。
>
> ——William Shakespeart

无论如何，我们和我们的公司，乃至于整个经济活动都存在于现实世界中，我们可以利用现实世界的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所有重要设计决定的集合，其中每个设计决定的重要程度则通过变更成本来衡量。
>
> ——Grady Booch

时间、金钱以及努力是软件架构中区分规模大小的衡量标准，也是架构和细节的区分标准。通过对这些要素的衡量，我们可以判断某个特定架构是好或坏：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，更需要在一段时间以内持续满足他们的后续要求。

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

一个系统的常规变更需求成本不应该是昂贵的，也不应该伴随着难以决策的大型设计而调整，更不应该需要一个独立的项目来单独完成。这些常规变更应该可以归入每日或者每周的日常系统维护中去。

要解决这个问题，还有一个不小的物理难题：时间旅行。我们怎么能够知道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间旅行机的情况

下，未卜先知，降低未来的变更成本呢？

> 所谓架构，就是你希望能在项目一开始就做对，但是却不一定能够做对的决策的集合。
>
> ——Ralph Johnson

了解历史已经够难了，我们对现实的掌握最多也只有七八成，预言未来就更加困难。

我们架构师们每天都站在这样拥有无穷选择的交叉路口。

其中一条比较黑暗的路线认为，只有威权和刚性才能带来强壮稳定的软件架构。如果某项变更成本太高，那么就忽视它——变更背后的需求要么被抑制，要么被丢到官僚主义的大机器中绞碎。架构师的决定是完整的、彻底极权的。软件架构就是全体开发人员的反乌托邦噩梦，永远是所有人沮丧的源泉。

而另外一条路线则充斥着大量投机性的通用设计。这会使软件项目中到处都是硬编码的猜测，无穷无尽的参数，或者成篇累牍的无效代码。维护这样的项目，常常遇到意外情况，预留多少资源都不够用。

本书试图探索的是一条极简优美的路线。这条路线认同软件的灵活多变性，并且保证这种灵活多变性仍是系统的一流属性。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决策。这条路线可以使人类的优势发挥作用，而非劣势。通过实际创造和探索，我们提出问题进行实验。优秀的架构一定是要靠一个优良的体系不断打磨才能产生的，并不是一成不变的。

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

遵循这条路线，我们需要用心、仔细观察，不停观察和思考，重视实践和原则。虽然这可能听起来很麻烦、很耗时，但是只有坚持走下去才能成功。

> 走得快的唯一方法，是先走好。
>
> ——Robert C. Martin

一起享受这个过程吧！

Kevlin Henney

2017 年 5 月

# 序

本书叫作《Clean Architecture：软件架构与设计匠艺》，使用这个名字可谓十分大胆，甚至可以说是自傲。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 岁的我写下人生的第一行代码起，到 2016 年，我编程已经超过 50 年。在这段时间里，我自认为学到了组织软件系统的一些方法——并且相信这些方法和经验对其他人有些价值。

我学习的方法是实际去构建一些大大小小的软件系统。我构建过小型的嵌入式系统，也构建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，也写过数据库类、数值计算类、几何计算类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，我获得了一个令人震惊的领悟：

软件架构的规则是相同的！

这个领悟的出乎预料之处在于，我所构建的这些系统之间差异巨大。为什么所有这些差异巨大的系统都遵守着同样的软件架构规则呢？我得出的结论是，软件架构规则和其他变量无关。

如果回顾一下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，它的 CPU 周期只有 0.5MHz，拥有 4KB 核心内存，32KB 磁盘存储，以及每秒只能传输 10 个字符的打字机接口。而现在，

我正在一辆游览南非的观光车上写这篇序言。我用的 MacBook 拥有 4 核 i7 处理器，每个核心 2.8GHz。这台电脑有 16GB 内存，1TB SSD 硬盘，可以用 2880×1800 虹膜显示屏展现高清视频。二者在计算能力上有着天壤之别。粗略分析即可知，这台 MacBook 至少比我半个世纪以前的计算机强大 $10^{22}$ 倍。

22 个数量级所带来的差距是非常非常巨大的，从地球到半人马星座也只有 $10^{22}$ 埃（angstrom，天文学单位），你口袋里的零钱所含的电子数量也差不多 $10^{22}$ 个。而这个数字（注意还是至少）是我在一生中，亲身经历的计算能力的提升。

既然计算能力发生了这么巨大的变化，那么其对于我所编写的软件影响有多大呢？软件大小当然变大了。我过去认为 2000 行的程序就很庞大了。毕竟这样的程序打成纸板能装满一盒子，重量超过 10 英磅。而现在，一个不超过 10 万行的程序，都不能算大程序了。

同时，在软件性能上当然也有所提升。我们现在可以轻轻松松地完成 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，但是都没那么成功。在这些电影中，获得了智能的巨型机器占据主导，而我们目前所拥有的依然只是计算机，虽然体积之小是当初难以想象的。

同时，还有一点很重要：今天的软件与过去的软件仍然是一样的。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言，以及更高级的编程范式。毕竟，我们现在用 Java、C#、Ruby 编写程序，并且大量采用面向对象编程方式。这没错，但是最终代码仍然只是顺序执行、分支语句、遍历的组合，这方面和 1960 年，甚至 1950 年的程序是一模一样的。

如果深入观察计算机编程的本质，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，无非编程语言稍微进步了一点，工具质量大大提升了。但是计算机程序的基本构造没有什么变化。

如果一个 1966 年的计算机程序员穿越时空到 2016 年，让她[1]在我的 Mackbook 上用 IntelliJ 写 Java 程序，她可能也就需要 24 小时来适应一下，她就能很快从头开始了。因为 Java 其实和 C 区别并不大，和 Fortran 也没那么大区别。

---

1 这里用她，因为当时计算机程序员大部分都是女性。

如果把你传送回 1966 年，告诉你如何在一个 10 个字符一秒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 小时来适应。毕竟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，导致软件架构的规则保持了一致。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，对它们进行排列组合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技术都是崭新的，是从来没有过的，过去的规则已经过时，不再适用。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，但软件架构的规则都仍然和 1946 年阿兰·图灵写下第一行机器代码的时候一样。

当然也有一点变化：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们，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，对这些规则有了一定程度的了解。

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。

# 致谢

下列这些人参与了本书的编纂，在此表示诚挚的谢意。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

还有许多其他人，这里不方便一一列出。

在本书的最后审校过程中，在我读第 21 章时，Jim Weirich 热情的微笑和富含旋律的笑声一直回响在我的脑海中。Jim，祝你一切顺利!

# 关于作者

Robert C. Martin（Bob 大叔）从 1970 年编程至今。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。同时，他还是 Bob 大叔咨询公司的创始人，该公司为全球大型公司提供软件开发咨询服务、培训以及技能培训服务。同时，他在 8th Light 公司任“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。本书作者在各种行业周刊上发表了十余篇文章，同时也经常被国际会议和行业峰会邀请进行演讲。他曾任 *C++ Report* 的主编，并且曾任敏捷联盟（Agile Aliance）的主席。

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*。

# 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。

- **下载资源**：本书如提供示例代码及资源文件，均可在 下载资源 处下载。
- **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/34261*

# 目录