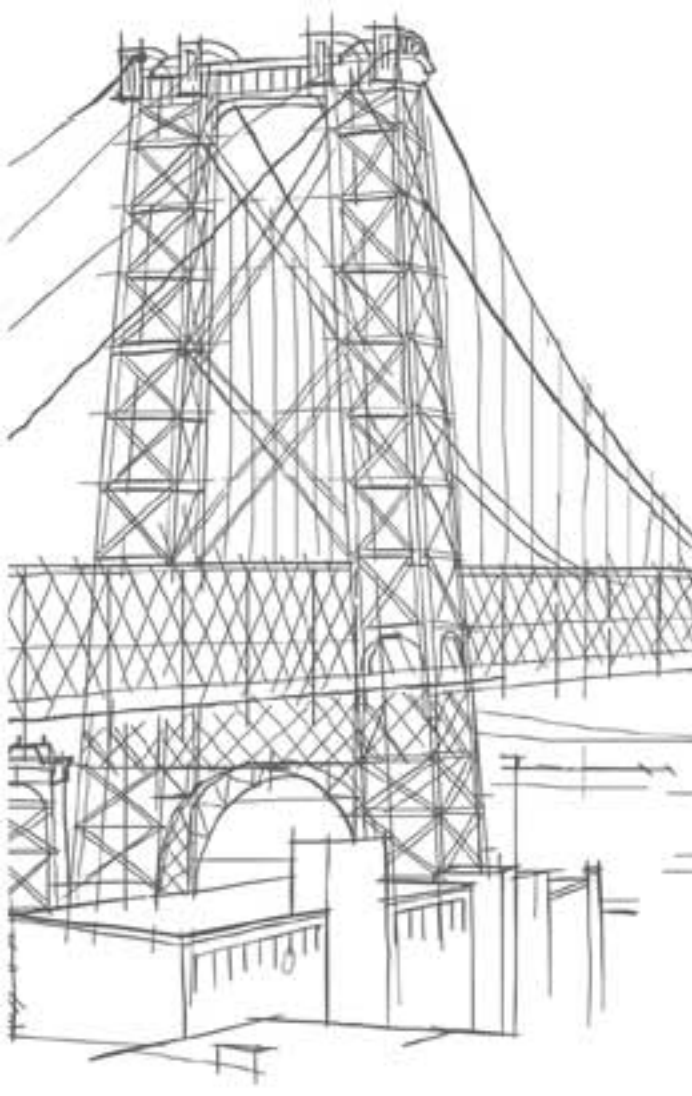
REACTIVE DESIGN PATTERNS

# 反应式设计模式

罗兰·库恩(Roland Kuhn)
[美] 布赖恩·哈纳菲(Brian Hanafee) 著
杰米·艾伦(Jamie Allen)

何品 邱嘉和 王石冲 译

林炜翔 审校



MANNING

清华大学出版社

# 反应式设计模式

罗兰·库恩(Roland Kuhn)
[美] 布赖恩·哈纳菲(Brian Hanafee) 著
杰米·艾伦(Jamie Allen)
何 品 邱嘉和 王石冲 译
林炜翔 审校

清华大学出版社
北 京

Roland Kuhn, Brian Hanafee, Jamie Allen
Reactive Design Patterns
EISBN: 978-1-61729-180-7
Original English language edition published by Manning Publications, 178 South Hill Drive, Westampton, NJ 08060 USA. 

北京市版权局著作权合同登记号　图字：01-2017-4218



**图书在版编目(CIP)数据**

反应式设计模式 / (美)罗兰 • 库恩(Roland Kuhn)，(美)布赖恩 • 哈纳菲(Brian Hanafee)，(美)杰米 • 艾伦(Jamie Allen) 著；何品，邱嘉和，王石冲 译. —北京：清华大学出版社，2019
书名原文：Reactive Design Patterns
ISBN 978-7-302-51714-6

Ⅰ. ①反…　Ⅱ. ①罗…　②布…　③杰…　④何…　⑤邱…　⑥王…　Ⅲ. ①软件开发　Ⅳ. ①TP311.52

中国版本图书馆 CIP 数据核字(2018)第 259605 号

**责任编辑：**王　军　韩宏志
**封面设计：**周晓亮
**版式设计：**思创景点
**责任校对：**牛艳敏
**责任印制：**丛怀宇

**出版发行：**清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　邮　　编：100084
社 总 机：010-62770175　　邮　　购：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
**印 装 者：**三河市铭诚印务有限公司
**经　　销：**全国新华书店
**开　　本：**170mm×240mm　　**印　　张：**24.75　　**字　　数：**471 千字
**版　　次：**2019 年 1 月第 1 版　　**印　　次：**2019 年 1 月第 1 次印刷
**定　　价：**98.00 元

产品编号：075684-01

# 致 Reactive Design Patterns 中文版读者

能看到《反应式设计模式》以其他语言出版，是我的荣幸；中文版的推出，将使中国的众多软件工程师以及系统架构师更便捷、更充分地获知书中内容。在见证了译者 Wayne Wang(王石冲)、Kerr(何品)、Hawstein(邱嘉和)以及他们的技术评审 Neo Lin(林炜翔)在翻译过程中投入的饱满热情以及对细节的极致追求后，更让我深感荣幸。亲爱的读者，你们此时手中所握，正是他们辛勤付出的结晶，而我也深受感动和启发。我衷心期望，本书能为你们开拓“论证回弹性分布式系统设计”的新思路，在全球人类文明互联的当下，能帮助你们继续挑战软件工程进步的极限！

Roland Kuhn 博士
《反应式设计模式》的主要作者

# Letter for the Chinese Translation of Reactive Design Patterns

It is a great pleasure to see *Reactive Design Patterns* translated into foreign languages, in particular for China where many software engineers and architects are not adequately reached by the English version. An even greater pleasure was to witness the enthusiasm and attention to detail put into this effort by 王石冲 (Wayne Wang), 何品(Kerr), 邱嘉和(Hawstein), and their technical editor 林炜翔(Neo Lin). I am very grateful and inspired by their hard work which you, dear reader, are now holding in your hands. May this book open up new ways of reasoning about resilient distributed systems for you and help you in continuing to push the envelope of software engineering in the globally connected age of human civilization.

Dr. Roland Kuhn

Main author of *Reactive Design Patterns*

# 译者简介

**何品**　热爱反应式编程，也是 Akka 和 Netty 等项目的贡献者，活跃于 Scala 社区，目前就职于淘宝。

**邱嘉和**　热爱编程与开源，Akka 贡献者，活跃于 Scala 社区，曾就职于豌豆荚与 GrowingIO，目前在创业中。

**王石冲** Scala 程序员，Lightbend 全家桶爱好者，反应式宣言践行者，目前在数云公司任架构师一职，从事流式数据引擎开发。

## 技术审校简介：

**林炜翔** 对软件系统工程化精益求精的中年程序员，曾参与多种业务场景下的软件开发。他对跨领域的软件系统设计有独到的见解。

# 译者序一

我从 2012 年开始接触 Netty，并随后了解到 Scala 和 Akka。随着对这些技术栈的深入学习，逐渐建立了自己的知识脉络和体系。源于对这些项目的热爱，贡献了一些代码，也结识了一些朋友。

我本来打算与嘉和写一本关于 Akka 的书，甚至已经设计好封面，不过当时由于 Akka 的前技术领导者主笔的这本书快要出版，嘉和就鼓动我翻译，我们一拍即合。不过本书的翻译难度颇高，涉及多个领域的相关知识，王石冲的加入为我们注入更大信心，而林炜翔一丝不苟的审校和陈涛的持续投入则极大地提高了本书的翻译质量。

本书围绕反应式宣言(https://www.reactivemanifesto.org/zh-CN)展开，讲述什么是反应式、为何需要反应式，以及反应式系统设计与开发中的一些常用模式，无论是软件开发者还是系统架构师，都可从本书中汲取知识养分。本书提出许多真知灼见，勾勒出反应式应用程序、反应式系统以及反应式平台等概念，有场景、有故事、有概念、有实践，令人沉醉其中，流连忘返。只可惜，受限于本书的篇幅，有些细节的讲解还不够全面透彻，还需要读者参考书中的指引自行研习。

由于篇幅所限，本书的主要例子都用 Scala 语言描述，运用了现成的 Akka 套件中提供的一些功能，但本书提及的概念、知识点、设计和模式并不局限于特定的语言和框架，读者可任意选择并亲手实践。以反应式流(https://github.com/reactive-streams/reactive-streams-jvm)为例，读者可选用 Akka-Stream、Reactor 或 RxJava 进行实践。读者也可多方参考、兼收并蓄，进而融会贯通。

作为本书的译者之一，我非常希望，本书可帮助读者构建出反应式系统设计的体系化知识结构，无论大家最终选用何种工具，都能得心应手地构建自己的反应式系统。当然，我也鼓励读者更多地参与到开源项目中，贡献自己的力量。作为本书的早期读者之一，我认为本书作为一本模式书已经足够，而实际开发对领域知识、编程技能以及架构能力都有较高的要求，需要读者进行更多的思考和实践。

我们将本书的相关源代码放到 GitHub，以方便读者下载，并提出问题和反馈意见。另外提供在线版代码清单。在技术审校林炜翔的帮助下，本书翻译质量得到极大提升；作为深耕电信、金融领域多年的外籍专家，他对一些英语文化梗的拿捏也

恰到好处。我们也得到了社区的极大帮助，特别是杨云、沈达、欧宁猫。而在反应式宣言的翻译上，更是得到了多方的帮助。

当然，最应该感谢的是我的爱人和两个女儿，感谢她们的体谅和支持，她们是我一切动力的来源。

何　品

2018 年 7 月于杭州

# 译者序二

我就读研究生期间就开始接触函数式编程，涉猎过几种函数式语言后，最终情定 Haskell。我对 Haskell 巧妙、灵活地解决问题的能力极为着迷，以至于参加工作成为一名 Java 程序员后，仍在利用业余时间学习 Haskell。再后来，受邓草原老师的影响，开始进入 Scala 世界，踏上学习、使用 Akka 进而为 Akka 项目贡献代码的道路。

Roland 做了多年 Akka 项目的技术领导，对反应式系统和 Akka 有深刻的思考和独到的见解。我在给 Akka 贡献代码的过程中得到其悉心指导，所以得知他正在编写一本关于“反应式系统”设计模式的书籍时，我第一时间在 Manning 上购买了其 MEAP 版本(Manning Early Access Program)，跟随 Roland 的指点，一点点地整理零散的架构模式和原则并形成系统。那时，我曾与何品讨论合写一本关于 Akka 的书籍。当本书英文版临近出版时，我向何品建议翻译本书。这不是一本讲述 Akka 的书，而是讲述反应式架构原则和设计模式，但这些原则和模式却在 Akka 发展过程中不断注入 Akka 中。可以说，Akka 被注入“反应式”灵魂。反过来，用 Akka 构建反应式系统是如此自然和简单。随着 Scala 社区的不断壮大，越来越多的人接触到 Akka，并开始用它构建应用程序。然而，我看到太多人还抱着旧思维模式，在不适当地使用 Akka，导致 Akka 的先进性无法发挥出来。正因为如此，我觉得更有必要翻译这本讲述反应式系统“内功心法”的书籍，以飨中文读者。

翻译书籍是一项艰辛的工作，翻译本书占用了我绝大部分业余时间。这本书最终是何品、石冲和我共同翻译完成的。没有他们，本书也就不可能在如此短的时间内完成。特别是何品，除了翻译书籍本身，还负责处理沟通和协调等事项。另外，特别感谢林炜翔和陈涛，他们怀着极大的热情对本书的译稿进行审校，进一步提高了本书的翻译质量。

当然，最应该感谢的是我的爱人，感谢她在本书翻译过程中给予的理解与支持。

邱嘉和

2018 年 7 月于北京

# 译者序三

我从 2013 年开始接触 Scala。别人因为畏惧 Scala 望而却步，我却因为觉得简单而欲罢不能。后来才领悟出，这是因为 Scala 是一门可以学习和实践的语言。我能通过理论学习迅速找到 Scala 以及基于它构建的框架的正确用法和最佳实践，而不需要通过“踩坑”来积累经验；我可以迅速地利用 Play 和 Akka 构建高性能、高可用的项目，并在生产中发挥作用。这里必须特别感谢数云公司以及技术总监韩铮对我的信任和“放纵”，使得我可以迅速积累 Play 和 Akka 实践经验，并逐渐整理出自己对使用 Actor 模型的心得。后来因为在何品的 Akka 群里逐渐变得活跃，并因而与他相识。也是机缘巧合之下，由于特别想感受创业公司氛围，我从数云离职，在北京的 GrowingIO 认识了嘉和，并和他一起经历了一段“一年抵三年”的岁月。在 2017 年 6 月，嘉和提议翻译，何品寻找到版权方时，我积极响应。之后，我们便开启了《反应式设计模式》的翻译之旅。

说实话，本书在 Akka 爱好者中可以说是万众期待。Akka 因为使用了 Actor 模型而闻名，但为什么使用 Actor 模型就能带来诸多好处呢？“放任崩溃”是好模式，但崩溃后怎么恢复呢？Actor 在分布式中是一把好手，但分布式下的 Actor 又有什么不同呢？然后，又如何基于 Actor 的消息发送机制得到 Akka Stream，这种流式处理又有哪些优点呢？还有，应该如何应对分布式数据复制、资源管理、流量控制、状态管理等不同场景？Akka 文档足够详尽，但很多功能的设计思想并没有阐释清楚，很容易让人落入只知其然不知其所以然的状态。

而本书能最好地解答你的几乎所有疑问。其中很多理论并不局限于 Akka，在构建高性能、低耦合、健壮的分布式系统中，也能大放异彩。作为还算资深的 Akka 使用者，我也是跟随本书的讲解，才真正弄懂 Akka 在设计上的许多精妙之处，以及反应式流的基本原理和 Akka Stream 的正确用法。这里必须感谢原作者们奉献了这本我们认为必成经典的书籍。

当然，不可否认的是，书里面仍有很多不足之处。有时读着会忽然发现意犹未尽，却戛然而止。我们曾就此询问 Roland 博士，他坦言当时因为篇幅和个人精力所限，很多地方未能深入展开。不过，大致方向已经列出来了。所以读者在阅读时，如果发现有些地方不够深入，可自行查找资料，以便更深入地理解和应用。

如果遇到任何问题，可以访问 Github 向我们提问；如果问题得到确认，我们会请你喝可乐。衷心希望所有阅读本书的读者，都可与我们共享编程乐趣！

在本书翻译过程中，需要特别感谢技术审校林炜翔、喜马拉雅 fm 的陈涛以及来自何品 Akka 群的各位帮忙评审的朋友，正是在他们的帮助和监督下，本书翻译质量才能得到保证。尤其是林炜翔，因为他加拿大华人的身份，帮助我们解读了很多也许只有土生土长的外国人才能理解的梗和俗语。希望有今后的工作中，我能把从翻译过程中学到的知识成功应用到公司的产品开发中，创造更大价值。

最后感谢我的妻子。你是我做一切事情的动力。

王石冲

2018 年 7 月于上海

# 技术审校序

在 2013 年，因为 Spark 初创团队在加拿大从事业务推广，我才了解到 Scala 以及其生态圈的存在，而那时的 Spark 只是蒙特利尔小众技术 meetup 里面的一个小话题。我并没有因为很早就接触 Spark 而成为大数据风口的追风人，反而在那间昏暗的地下室里看到了 Scala 带来的无限潜力。作为一个中年人，一个资深编程人员，在代码质量和业务效率的平衡上，我一直有自己独到见解和追求，而且这一路走来，其实累积了不少疑惑。而 Scala 及其充满活力的社区为我敞开了一扇新的大门，得以验证我长期以来的一些想法，以便更好地完善和沉淀自己的知识体系。

在寻找这些答案的路途上，我甚至完全偏离了自己原先的职业轨迹，投身 IT 领域做电商开发，再甚至从加拿大回到中国，尝试一些分布式、纯函数式编程的高级商业应用实践。

至于参与本书的技术审校，也是在这条道路上的一段加速小跑。本书尝试建立(尚需完善)的理论体系基础，是原作者多年来理论和实践的升华，他希望通过把这些认知提炼为落到纸上的文字，以便为追求工程质量的同行助力。然而对于大部分读者而言，原版书中有不少技术名词或理论显得相对生僻，一些细心的读者甚至在一些中文名词的选用上另有看法，对于这一点，我希望读者多花点时间通读全书，再自行判断；希望读者能在这个过程中，审视、回顾甚至推翻自己的某些知识结构，构建出更强大的知识体系。毕竟，每个软件工程师都有改变世界的梦想，可我们很多时候只是获得了改变旧世界的工具，但缺少一个构造新世界的蓝图。

希望本书的面世，能成为广大软件工程工作者勾勒各个宏大蓝图时的有益借鉴。

最后，感谢远方的家人，对我的任性给予理解、包容和支持。

林炜翔

2018 年 7 月于北京

# 序　言

非常感谢 Roland 耗费心血撰写这本基础书籍，我实在想不出还有谁更有能力胜任这项工作。Roland 是一个思维敏锐、知识渊博的人。Roland 与其他人合著了《反应式宣言》(*Reactive Manifesto*)，并多年来一直是 Akka 项目的技术领导。他还参与编辑和讲授了 Coursera 平台上广受欢迎的“反应式编程和设计”课程，是我遇到过的最优秀的技术作家。

诚然，我对本书的出版感到兴奋。本书概述了反应式架构/设计的含义，并且务实、透彻地分析了反应式的各种第一原理/公理。此外，本书也给出了反应式设计模式的分类目录(catalog)，如何构思系统设计，以及这些模式之间的整体关联——就如 Martin Fowler 的《企业应用程序架构》在 15 年前所做的一样。

在自己的职业生涯中，我亲眼看到了弹性、松耦合、消息驱动的系统可带来巨大效益，与那些隐藏了分布式系统本质的传统方案相比更甚。2013 年左右，我想将这些经验教训规范化，《反应式宣言》应运而生。我记得，《反应式宣言》最初只是我在 Typesafe 公司(现在的 Lightbend 公司)的内部技术会议上分享的一些草稿。恰巧，这次技术会议和 Scala Days New York 同场，期间又遇上了 Roland、Martin Odersky 和 Erik Meijer，他们在那里拍摄了他们那个“不专业”但又蛮有趣的反应式编程 Coursera 课程宣传片。宣言中基于各项反应式原则的探讨引起了其他工程师的共鸣，在 2013 年 7 月发布了初版。此后，该宣言从开发社区收到了大量反馈意见。Roland、Martin Thompson、Dave Farley 和我对该宣言进行了大量改进和修正，并最终在 2014 年 9 月发布了第 2 版。到目前为止，已有超过 2.3 万人签署了这个宣言。在此期间，我看到“反应式”进展显著，从一项几乎不被承认的、只在少数公司的一些边缘系统中使用的技术，发展为众多大公司的整体平台战略，涵盖中间件、金融服务、零售、社交媒体、彩票以及游戏等各个不同领域。

《反应式宣言》将“反应式系统(Reactive Systems)”定义为一系列架构设计原则，旨在解决系统当前和未来面临的挑战。这些原则也非首开先河；可追溯到 20 世纪 70 年代和 80 年代由 Jim Gray 和 Pat Helland 在 Tandem System，以及 Joe Armstrong 和 Robert Virding 在 Erlang(及 OTP)上所做的开创性工作。然而，这些先驱者都领先于他们所处的年代，直到过去五年，科技产业才被迫重新思考当前企业软件开发的最佳做法，并学会将得来不易的反应式知识框架应用到当今的多

CPU 核心架构、云计算以及 IoT 领域。

至今，反应式的多项原则已对行业产生巨大影响；与许多成功想法一样，它们被过度使用和重新诠释。这不纯粹是坏事；思想需要不断进化，从而保持相关性。但这也会引起混淆，导致原始意图被淡化。例如，一种不幸的流行误解是：反应式编程无非就是使用回调或面向流的组合子进行异步和非阻塞的编程(这些技术归结成反应式编程)。如果仅关注这个角度，就意味着将错过反应式各项原则的其他许多好处。本书的贡献之一便是从更宏大的视角(系统视图)将焦点从一个模块如何独立提供服务，聚焦到如何设计一个具有协作性、回弹性和弹性的系统：反应式系统。

与 *Design Patterns: Elements of Reusable Object-Oriented Software* 和 *Domain-Driven Design* 一样，这本明日经典书籍也是每位专业程序员的案边必备。预祝你享受到阅读和学习的乐趣！

——Jonas Bonér

Lightbend 公司 CTO 和创始人

Akka 创始人

# 自　序

早在正式加入 Akka 团队前，Manning 出版社的 Mike Stephens 就试图说服我写一本关于 Akka 的书籍。我很想说：“好啊！”，但我当时正要变动工作和国籍，我的妻子也提醒我：写一本书需要付出大量心血。但此后，写一本书的想法便在我心中扎根了。又是三年，在《反应式宣言》发表后，Martin Odersky、Erik Meijer 和我在 Coursera 平台上讲授 Principles of Reactive Programming 这门公开课程，期间参与学习的学生逾 12 万人。这门课程的最初想法来自于 Typesafe 的一次开发者会议，在那次会议上，我向 Martin 建议，我们应通过演示如何在规避陷阱的同时高效地使用这些反应式工具，来促进反应式编程的蓬勃发展——结合我自己在 Akka 的邮件列表上答疑的经验，我知道人们通常都会有哪些疑惑。

视频课程效果不错，覆盖面广，和学生们在讨论组互动，能普遍改善大家的“生活水准”。不幸的是，由于受形式上的限制，在这个主题上进行的讨论的深度和广度都是有限的。毕竟在七周时间里，只能展示那么多内容。因此，我还是渴望写一本书来传达我对反应式系统(Reactive system)的思考。如果只描述 Akka，内容将过于简单，我觉得，要是我写一本书，那么它应涵盖更大范围。我喜欢研究 Akka，它确实改变了我的生活，但 Akka 仅是一种表达分布式和高可靠系统的工具，并非所需要的唯一工具。

于是，我就开始了你手中这部作品的创作之旅，这项任务十分艰巨，我知道我需要助力。幸运的是，那时 Jamie 刚完成他的 *Effective Akka*，所以立刻加入了创作队伍。可白天写书对我们来说太奢侈了，导致本书的启动过程很慢，并且一直都在延期。原计划在 Principles of Reactive Programming 课程的第一次迭代期间就将前三章内容放在网上，并开启早期预览计划，可最终不得不延后数月。令我们惊讶的是，即使我们知道某个观点主要写哪些内容，可当将心中的想法真正转化成文字时，却发现缺少很多细节。随着时间的推移，Jamie 的日常工作越来越繁忙，不得不完全停止参与创作。再后来，Manning 的技术开发编辑 Brian 参与到了这个项目，很快变得很明显的就是：他不仅提出了非常好的建议，并且以身作则，所以也将他作为合著者之一写在封面上。最终，我在 Brain 的帮

助下完成了本书手稿的撰写。

本书不仅包含关于何时以及如何使用反应式编程以及相关工具的建议，也解释背后的缘由；这样你就可以根据自己的需要做适当的调整，从而满足不同的需求以及新的应用场景。我希望本书能激励你更多地学习，并进一步探索“反应式系统”的奇妙世界。

Roland Kuhn

# 作者简介

Roland Kuhn 博士曾在慕尼黑工业大学学习物理专业，获得了博士学位；在欧洲核子研究中心(瑞士日内瓦)的高能粒子物理实验中，发表了关于核子的胶子自旋结构测量的博士专题论文。该实验需要使用和实现大型计算集群以及快速的数据处理网络，这也为 Roland 透彻理解分布式计算奠定了基础。此后，Roland 博士在德国空间运营中心工作了 4 年，负责建设军事卫星的控制中心和地面基础设施。再后来，他加入 Lightbend(之前叫做 Typesafe)公司，在 2012 年 11 月到 2016 年 3 月期间负责带领 Akka 团队。在此期间，他与 Martin Odersky 和 Erik Meijer 一起在 Coursera 平台上讲授 Principles of Reactive Programming 课程，这门课程的学员超过 12 万人。Roland 与 Jonas Bonér 等人共同撰写了第一版的《反应式宣言》，该宣言于 2013 年 6 月发表。目前，Roland 是 Actyx 的首席技术官及联合创始人，Actyx 是一家总部位于慕尼黑的公司，致力于使欧洲的各类中小型制造企业享受到现代反应式系统的福泽。

Brian Hanafee 在加利福尼亚大学伯克利分校获得电气工程与计算机科学学士学位，现任富国银行的首席系统架构师，负责设计网上银行和支付系统，并长期引领公司的技术门槛提升。此前，Brian 曾在甲骨文公司工作，致力于研究新兴产品、互动电视系统以及文本处理系统。Brian 也曾任博思艾伦咨询公司的咨询师，并曾在 ADS 公司将人工智能技术应用到军事规划系统中。Brian 还为第一代弹射安全的头盔综合显示系统编写了软件。

Jamie Allen 是星巴克 UCP 项目的技术总监，致力于以跨运营模式、跨地域的方式，为星巴克公司各地的消费者重新定义数字体验。他是 *Effective Akka* 一书的作者，曾与 Roland 和 Jonas 一起在 Lightbend 公司工作 4 年以上。Jamie 自 2008 年以来一直从事 Scala 和 Actor 开发工作，与世界各地的客户合作，帮助他们理解和采用反应式系统设计。

# 致　谢

首先感谢 Jamie，没有他，我就不敢接手这个项目。但最深切的感激还是属于 Jonas Bonér，他创立了 Akka，并委以我引领 Akka 发展的重任，一路走来都支持着我。也深深感谢 Viktor Klang，他无数次就生活和分布式系统话题与我进行过严肃探讨，教我如何以身作则、迎难而上。还要特别感谢 Jonas、Viktor 和 Patrik Nordwall，在我休假三个月专注于创作本书期间，他们替我站好了 Akka 技术领导岗。感激 Brian 和 Jamie 和我一同挑起重担，和这些值得信赖的同伴一起工作，令我感到欣慰和鼓舞。

感谢 Sean Walsh、Duncan DeVore 以及 Bert Bates 审阅早期手稿，并帮助我拟定了如何表述各种反应式设计模式的基本结构。还要感谢 Endre Varga 耗费精力为 Principles of Reactive Programming 课程设计 KVStore 习题——这构成了本书第 13 章使用的状态复制示例代码的基础。感谢 Pablo Medina 帮我组织 13.2 节的 CKite 例子；同时感谢技术校对 Thomas Lockney，他始终能敏锐地找到正文中的错误。感谢以下独立评审者的慷慨奉献：Joel Kotarski、Valentine Sinitsyn、Mark Elston、Miguel Eduardo Gil Biraud、William E. Wheeler、Jonathan Freeman、Franco Bulgarelli、Bryan Gilbert、Carlos Curotto、Andy Hicks、William Chan、Jacek Sokulski、Christian Bridge-Harrington 博士、Satadru Roy、Richard Jepps、Sorbo Bagchi、Nenko Tabakov、Martin Anlauf、Kolja Dummann、Gordon Fische、Sebastien Boisver 以及 Henrik Lovborg。真诚感谢热心的 Akka 社区以及所提供的宝贵素材，帮助我们加深了对分布式系统的理解。

感谢 Manning 出版社负责本书的团队，是他们让本书成为可能。尤其要感谢 Mike Stephens，他一直唠叨到我答应写作本书才作罢，Jenny Stout 则不断督促我前进，还要感谢 Candace Gillhoolley 为本书推广所付出的努力，我想要单独说明的是，Ben Kovitz 是一位极其认真负责的排版编辑，感谢 Tiffany Taylor 和 Katie Tennant 对表意不明的段落进行润色。

最后，以所有读者之名，我向妻子 Alex 致以最大的谢意与爱。她怀着极大的怜悯之情，忍受了我无数小时的精神陪伴缺失。

——Roland Kuhn

感谢我的妻子 Yeon 以及我的三个孩子 Sophie、Layla 和 James。同样非常感

谢 Roland 邀请我参与本书的创作；也要感谢 Brian，感谢他推动本书的最终出版，并贡献了他的专业知识。

——Jamie Allen

感谢妻子 Patty 一贯的支持，感谢我的两个女儿 Yvonne 和 Barbara，感谢她们帮我回顾《神秘博士》的剧情，并“口是心非地”称赞我讲的笑话十分有趣。感谢 Susan Conant 和 Bert Bates，感谢你们让我加入，并教导我如何以书本形式传道授业解惑。最后，感谢 Roland 和 Jamie，感谢你们向我展示了反应式设计的各项原则，并欢迎我加入本书的创作中来。

——Brain Hanafee

# 前　言

本书旨在成为引导你理解和设计反应式系统的综合性指南，不仅提供《反应式宣言》的注解版本，还包括开创该宣言的缘由和论据。本书浓墨重彩地描述一些反应式设计模式，这些模式实现反应式系统设计的多个方面；还列出了更深层次的文献资源，以便你进一步研究。所陈述的模式形成一个连贯整体，虽然并非详尽无遗，但其所包含的背景知识将使得读者能在需要的时候识别、提炼和呈现出新模式。

## 读者对象

本书面向每一位想要实现反应式系统的人士。

- 本书涵盖反应式系统的架构设计以及设计理念，向架构师简要介绍反应式应用程序及其组件的特性，并讨论了这些模式的适用性。
- 践行者将受益于书中对于每个模式所适用场景的详尽讨论。本书列出各模式的应用步骤，并配备完整的源代码；讲述了在不同场景下，如何灵活运用和适配这些模式。
- 希望学到更多知识的读者在观看了 Principles of Reactive Programming 视频课程后，将乐意了解反应式原则背后的思考过程，并可遵循参考文献做进一步的研究。

阅读本书前，读者不必预先了解反应式系统，但仍然需要熟悉通常的软件开发，并具有一些排除分布式系统引发的困难的经验。对于某些部分，基本理解函数式编程将有所裨益(例如了解如何使用不可变值和纯函数进行编程)，但不必了解范畴论。

## 导读

本书的内容是特意组织编排的，以便读者可像读一本故事书那样翻阅。首先呈现一个介绍性示例，概述《反应式宣言》以及反应式工具集，进而探讨反应式原则背后的哲学，最后从不同角度阐述设计反应式系统所需的设计模式。这段旅程涵盖大量知识领域，并在文字描述中引用了不少额外的背景资料。通读一遍，

浏览相关内容，你将建立对书中知识范围的直觉。但这通常只是进一步研究的起点；在自己的项目中应用从本书学到的知识时，可回头再次研读，那时会获得更深刻的洞察力。

如果你已经熟悉反应式系统面临的挑战，可跳过第 1 章；如果你已经熟悉大多数行业主流工具，则可跳过第 3 章。时间紧迫的读者可直接开始阅读第 III 部分讲述的各个模式，但依然建议首先学习第 II 部分：模式的描述过程常引用相关解释及理论背景，这些内容都是对应设计模式的衍生基础。

具有更多设计和实现反应式系统的经验后，预计你将再次研读那些更富哲理性的章节——尤其是第 8 章和第 9 章；首次阅读时，会觉得这两章的内容难以理解，请不必担心，反复研读即可。

## 约定

由于在作为编程概念时，对英文单词 future 的多重解读已严重偏离其本身的含义，因此，所有将其作为编程概念引用的地方都使用首字母大写的 Future，就算没有出现在代码字体中也是如此。

英文单词 Actor 的情况略有不同，在日常英语中 Actor 指舞台上的一个人，以及一个动作或处理过程的参与者。因此，这个单词只有在特别指 Actor 模型，或在代码字体中作为 Actor 特质出现时，才会大写。

## 源代码下载

本书的示例源代码(作者提供的源代码)可从 GitHub 下载：https://github.com/ReactiveDesignPatterns/CodeSamples/。

本书译者对源代码进行了重新调试，新代码下载位置如下：https://github.com/ReactivePlatform/Reactive-Design-Patterns。

本书正文列出的代码都是经过译者重新调试过的代码。

读者也可扫描封底的二维码，下载这两套代码。

中文版还提供在线资源：https://rdp.reactiveplatform.xyz/。

GitHub 还提供其他功能，允许你讨论本书中的示例或报告问题，并欢迎你提出改进意见，这样，其他读者将受益于你的思考和经验。

书中大部分示例代码都用 Java 或 Scala 编写，并将 SBT 用作构建工具。要查阅 SBT 详细文档，请访问 https://www.scala-sbt.org/；要查阅 SBT 入门资料，可访问 https://github.com/ReactivePlatform/Notes/issues/8。为构建和运行示例代码，还需要使用 JDK 8。

## 其他在线资源

可访问 https://www.reactivedesignpatterns.com/获取本书所介绍模式的概述和附加材料。此外，读者可免费访问 Manning 出版社的私有 Web 论坛，在那里，可评论本书、提出技术问题，还可获得作者和其他用户的帮助。可用 Web 浏览器访问 https://www.manning.com/books/reactive-design-patterns。可从这个页面了解以下信息：注册后如何访问该论坛、可获得哪些帮助以及该论坛的一些行为准则。

Manning 承诺为读者提供一个交流场所，在那里，你可与作者以及其他读者进行有意义的对话。但作者不对参与程度做任何承诺，作者对 AO 的贡献仍是自愿的和无偿的。我们建议你向作者提一些富有挑战性的问题，以引起他们的兴趣！

只要本书英文版尚未绝版，就可从 Manning 出版社的网站上访问到作者在线论坛以及之前讨论的存档。

# 目　录

## 第II部分 微言大义



## 第III部分 设计模式

# 第Ⅰ部分

# 简　介

你曾思考过高性能Web应用程序是如何实现的吗？社交网络和大型零售网站肯定有一些秘密配方使得系统运行迅速并且可靠，但这些秘密是什么呢？本书将为你揭晓谜底，你将学习到这些近似永远不出故障[1]，并能满足数十亿人需求的系统背后的设计原则与模式。虽然你构建的系统未必有如此雄心勃勃的要求，但是它们的主要特质应该是一致的：

- 你想要你的应用程序可靠地工作，即使某些部件(硬件或者软件)有可能出现故障。
- 你希望你的应用程序在你需要支撑更多用户时，可持续提供服务，而且你希望能通过添加或者删除资源来调整它的能力[2]，从而适应不断变化的需求(没有可预测未来的水晶球的帮助，很难进行正确的容量规划)。

在第 1 章中，我们将勾勒出一个具备这些特质的应用程序的开发过程。我们将指明你会遇到的挑战，并基于一个具体例子(一个假想中的 Gmail 服务实现)给出解决方案，但我们将以不提供具体技术选型的形式进行[3]。

这个使用场景为接下来在第 2 章中对《反应式宣言》所进行的详细讨论作了铺垫。该宣言以简洁、抽象的形式撰写，目的是为了聚焦于它的本质：将多个独立的、有效的程序特性凝聚为一个整体，从而形成更大的合力。我们将通过把高度抽象的特质分解为更小的部分，并解释各部分又如何重新合为一体，来展现这一点。

---

1 这里指理论层面上的高可用性。在真实场景中，你仍然可能遇到这些系统出现故障的时候。——译者注

2 一般都利用公有云的能力，或者混合部署公有云和私有云，从而根据需求对所使用的资源进行动态伸缩。——译者注

3 即使用通用的模式，而非绑定到某种具体的技术实现。——译者注

第 3 章是本部分的最后一章，该章大致介绍了行业工具：函数式编程、Future、Promise、通信顺序进程(Communicating Sequential Processes，CSP)、Observer 和 Observable(Reactive Extensions)以及 Actor 模型。

# 第1章

# 为什么需要反应式?

我们的初衷是构建一个对用户即时响应的(responsive)系统。这意味着该系统无论在什么情况下，都能即时响应用户的输入。由于任何单台计算机在任何时刻都可能宕机，因此我们需要将这个系统分布到多台计算机上。引入分布式结构这个额外的基础需求使我们意识到：构建这样的系统需要新的架构模式(或者重新发现旧模式)。过去，我们建立了各种方法来维持某种表象：单线程的本地运算能够魔法般地扩展运行在多个处理器核心或网络节点上。然而，虚实之间的沟壑已经大到难以为继[1]。解决方法是让我们的应用程序中所具有的分布和并发的本质明明白白地反映到编程模型上来，并使其变成我们的优势。

本书将教你如何编写一种无论在部分组件宕机、程序运行失败、负载变化，甚至代码里有 bug 时，仍然能保持即时响应性(responsive)的系统。你将看到，这会要求你调整思考和设计应用程序的方式。下面是《反应式宣言》(Reactive Manifesto)[2]的四个信条，这些信条定义了一套通用词汇，并罗列了现代计算机系统面临的基本挑战。

1 例如，Java EE 服务允许我们透明地调用远程服务，这些服务其实自动连接在一起，其中甚至可能包括分布式数据库事务。网络失败或者远程服务过载等细节都完全被隐藏并抽象掉了。因为无法接触到这些内容，也导致开发者们无法在开发过程中进行周全的考量。

2 参见：http://reactivemanifesto.org。

- 必须对用户作出反应(即时响应，英文为 responsive[3])；
- 必须对失败作出反应，并保持可用性(回弹性，英文为 resilient[4])；
- 必须对不同的负载情况作出反应(弹性，英文为 elastic)；
- 必须对输入作出反应(消息驱动，英文为 message-driven)。

除此之外，创建系统时，如果脑海里带着这些原则，将指引你更好地完成模块化设计，无论是运行时的部署，还是代码结构本身。因此，我们在反应式的增益清单里面添加两个新属性：可维护(maintainability)和可扩展(extensibility)。图 1-1 以另一种方式展现了这些属性。

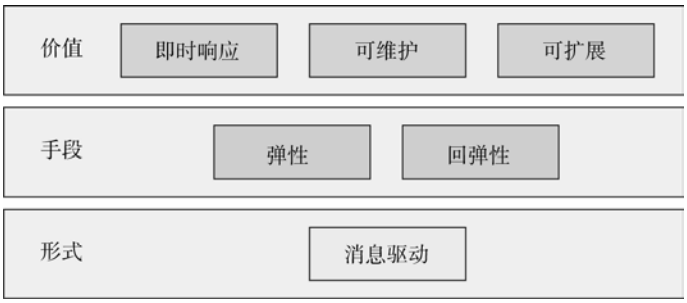



图 1-1 反应式的价值结构

在接下来的章节中，你将深入领悟《反应式宣言》蕴含的逻辑，并了解几个行业主流的工具以及蕴藏在工具背后的设计哲学，使你能够行之有效地运用这些工具实现反应式设计。由这些工具引出的设计模式将在本书第Ⅲ部分呈现。在尽力呈现宣言之前，我们先来探索创建反应式应用(reactive application)所面临的挑战，并用一个众所周知的邮件服务作为例子：设想如何重新实现 Gmail。

3 按牛津英英词典，此处英文的 responsive 有两种意思，一种是 reacting quickly and positively，即积极、迅速地反应；另一种是 answering，即应答的、反应的。系统正常运行时，responsive 应当取前者；在系统失败或者过载的情形下，则应该保障后者(参见 16 章流量控制模式里的丢弃模式)。读者需要分清这两种情境下 responsive 的不同意义。但整体来说，为强调反应式应用的迅速应答特点，我们将其翻译成“即时响应”。在附录 C 中，读者可以看到更具体的解释。——译者注

4 resilient 在之前被很多人翻译成“弹性”，这容易产生歧义，而且容易和 elastic 混淆。所以把 resilient 翻译成弹性的人又把 elastic 翻译成“可伸缩性”，但是又与 scalable 混淆，所以 scalable 翻译成“可扩展”，但是又和 extensible 混淆，然后他们再也找不出其他词了，所以 extensible 还是叫“可扩展”……实际上这个词更多是强调受压的时候回弹的那种弹性，强调有复原力、有抵抗力，所以本书翻译成“回弹性”。后续的几个词也都往前移动一位，回到它们的本意。——译者注

## 1.1　剖析反应式应用

要开始这样一个项目，首要任务是描绘出部署的架构图，并草拟出需要开发的软件模块清单。这也许不会是最终架构，但是你需要描绘问题空间，并探寻潜在的难点。我们将通过列举应用程序中的几个概括性功能来展开这个“重新实现”的 Gmail 例子：

- 应用必须提供一个视图，供用户查看各自的邮箱并展现相关内容；
- 为此，系统必须存储所有邮件，并保证它们可以被随时访问到；
- 系统必须支持用户编写和发送邮件；
- 为方便用户使用，系统应该提供邮件联系人列表，并支持用户管理联系人。
- 系统必须有一个好用的邮件搜索功能。

真正的 Gmail 应用程序有更多功能，但是上面的清单对于该例来说已经够用了。这些功能的某些部分之间的关系比其他部分更紧密：例如，展示和编写邮件都是用户界面的一部分，它们分享(或者竞争)同一块屏幕区域，而邮件存储的实现，则和这两点相对疏离些。搜索功能的实现则更贴近存储端而不是前端。

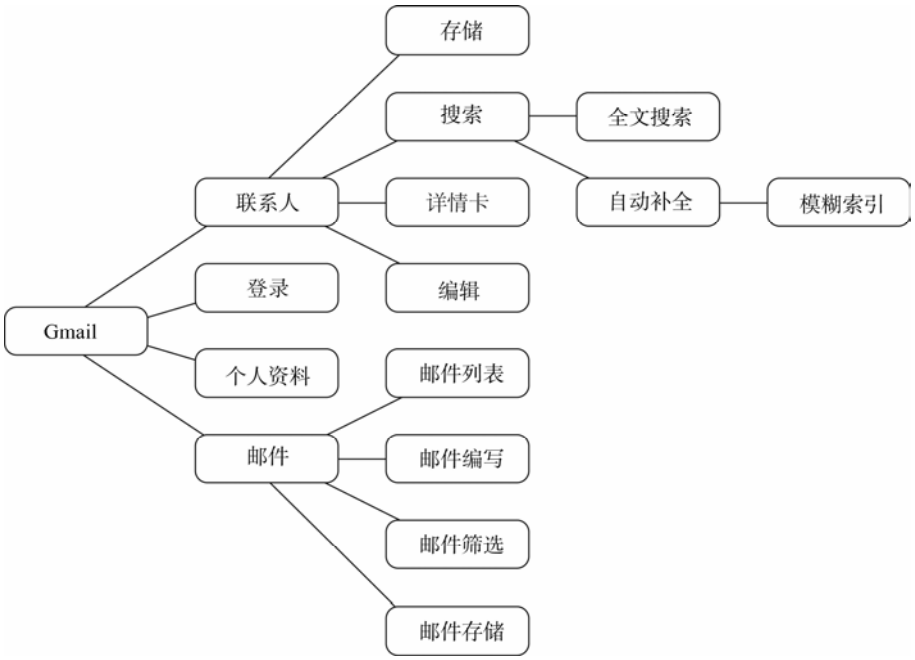



图 1-2　虚构的 Gmail 系统的部分分解模块层次结构图

这些考量点指引了 Gmail 的层次化拆分，将它的整体功能拆分成越来越小的部分。更准确地说，你可以应用第 12 章描述的“简单组件模式(Simple Component pattern)”，以确保清晰地划定和分离整个系统中的不同职责。“错误内核模式(Error Kernel pattern)”和“放任崩溃模式(Let-It-Crash pattern)”则是对这个过程的补充，以保证应用架构具备可靠的失败处理机制(failure handling)——不仅要防止机器或者网络故障，也要提防源代码里面没有被正确处理的、罕见的失败条件(failure conditions)，即 bug。

这个过程的结果将会是一套层次分明的组件，等着被开发和部署。图 1-2 展示了一个例子。其中，每个组件就其功能而言可能是复杂的，例如搜索算法的实现；也可能就其部署和编排过程而言是复杂的，例如为数十亿用户提供邮件存储。但在描述组件的个体职责时，总是应该保持简洁。

## 1.2 应对负载

存储所有邮件所需的资源将是巨大的：数亿拥有 GB 级邮件的用户的数据将需要 EB(exabytes)级别的存储容量。这样量级的持久化存储将需要由许多分布式机器来提供。任何单台存储设备都无法提供如此大的空间，而且将所有数据存放在同一个地方也是不明智的。分布式使得数据集对于局部危险(如自然灾害等)具有更强的抵抗力；但更重要的是，它使数据能够在更广阔的地域内被高效地访问到。如果用户群体是全球范围的，那么数据也应该在全球分布。将日本用户的邮件存储在日本或者临近日本的地方是更可取的(假设该用户大多数时间都是从日本登录的)。

上述洞察将我们引领到将在第 17 章描述的“分片模式(sharding pattern)”：你可将整个数据集分割成很多小片——即“分片(shards)”——并随后分而布之。因为分片的数量远小于用户的数量，所以让整个系统都知道每一个分片的位置是实用的。为了找到一个用户的邮箱，你只需要确定这个用户的数据属于哪一个分片就好。你可以通过给每个用户分配一个体现了地理密切关系的 ID(例如，使用开头几位数字来表示所在国家)来实现这个目标，之后这些 ID 就能直接根据数字被划分进正确的分片号里(比如，0 号分片可以包含 0~999 999 的 ID，1 号分片可以包含 1 000 000~1 999 999 的 ID，以此类推)。

这里的关键在于：数据集天然地由众多独立小片组成，每片彼此分离非常简单。对于一个邮箱的操作永远不会直接影响到另一个邮箱，所以分片之间并不需要进行沟通。每一个分片只为解决方案中的特定部分服务。

Gmail 应用另一个需要占用很多资源的部分是向用户展示文件夹和邮件。想

用中心化处理方式提供这项功能几乎是不可能的。这不仅仅有延迟的原因(即便是以光速，在全球范围内发送信息也会消耗可观的时间)，还有上百万用户每秒进行的海量交互次数。所以，你也需要在众多机器之间拆分任务。先从用户的电脑说起：绝大多数图形化展现都是由浏览器渲染的，从而将工作负载转移到非常靠近需要它的地方，并且实际上为每个用户分片了这些展现。

Web 浏览器需要从服务器获取原始信息，最理想的当然就是从最近的服务器上获取，以尽量缩短网络往返时间。连接用户与邮箱以及传递对应的请求与响应的任务也可被很容易地分片。在这个场景里，浏览器的网络地址直接提供了所有需要的特征数据，其中包括大致的地理位置。

一个值得注意的地方是，在前面提到的所有场景里，都可通过将分片变小，并将负载安排到更多机器上的方式来添加资源。机器的总量由用户或被使用的网络地址的数量决定，以这个数字来提供所需资源将是绰绰有余的。这个方案只有在服务单个用户需要的计算能力超过单个计算机所能提供上限时才需要调整，而这个时候，用户的数据集或者计算问题需要被拆分成更小的单元来处理。

这意味着通过将系统拆分成可分布的部分，你获得了扩展服务容量的能力，能通过使用更大数量的分片来服务更多用户。只要分片彼此独立互不依赖，那么系统在理论上就能无限扩展。当然在实际应用中，一个世界范围内的、拥有数百万节点的部署规模的系统的编排和维护需要耗费大量精力，并且必须有值得这样付出的价值。

## 1.3　应对失败

数据集或者计算资源的分片解决了为一般场景提供足够资源进行服务的问题。这时所有服务都平滑运行、网络也正常运转。但是为了应对失败，你还需要在意外发生时仍然保持运行的能力：

- 机器可能存在临时(例如发生了机器过热或者内核错误)或者永久(电力或机械故障、火灾、洪水等)故障。
- 无论是在计算中心里还是在互联网上，网络组件都可能发生故障——包括洲际跨海电缆连接中断这种情形造成的失联的网络分区。
- 运维人员或者自动维护脚本也可能会意外地损毁部分数据。

解决这个问题的唯一办法是在不同的位置复制系统的数据或者功能。副本放置的地理位置需要与系统的适用范围相匹配；例如全球化的邮件服务应该能够服务来自多个国家的每个用户。

相对于分片来说，复制是一个更加困难和多变的话题。虽然从直觉上来说，你不过是想在多个地方拥有相同的数据——但是让副本像我们所期待的那样保持同步不仅要付出很高的成本，还要做许多艰难的决定。比如说，如果远程副本暂时无法同步，那么我们是否应该使对于距离最近的副本的写入操作失败，或者延迟到远程副本可用时？如果最近副本已经发出了操作完成的信号，那么远程副本的尚未同步的老数据是否不应该被访问到？或者这样的不一致性出现的机会不大或者很短暂？在不同项目里，甚至一个特定系统的不同模块中，对于这些问题的答案都可能不尽相同。因此，将呈现一系列解决方案，这些方案使得你可以基于运维复杂性、性能、可用性和一致性进行权衡。

我们会在第 13 章中讨论几种能处理大部分上述特征的方案。以下是几个基本选项：

- 主动-被动复制(active-passive replication)——多个副本会商定出它们中的哪一个可以接受更新。当主动副本不再响应了，失败切换到哪一个副本需要剩余被动副本之间达成共识。
- 基于共识的多主复制(consensus-based multiple-master replication)——每一次更新操作都要被足够多的副本同意以获得跨所有副本的一致性行为，代价则是可用性和延迟。
- 带有冲突检测和解决方案的乐观复制(optimistic replication with conflict detection and resolution)——多个主动副本传播更新，并在冲突时回滚事务，或者丢弃在网络分区期间被执行的冲突更新。
- 无冲突的可复制数据类型(conflict-free replicated data types)——这种方法预先规定了合并策略，所以从定义上来说是不会产生冲突的。而代价则是只能提供最终一致性，并且在创建数据模型时需要特别的处理。

在 Gmail 例子中，有几个服务需要向用户提供一致性：如果一个用户成功地将一封邮件移动到另一个文件夹，那么无论该用户之后从哪个客户端访问他的邮箱，他都应该能在对应的文件夹里面看到这封邮件。联系人的电话或者用户资料的变更也都应该达到同样的效果。对于这些数据，你可以使用*主动-被动复制*的方式，并通过粗粒化的失败响应动作(即在副本范围内采取相同的失败响应方式)来保持系统的简单性。或者你也可以在单个用户不会并发地对相同的数据项进行带有冲突的变更的假设下，采用“乐观复制”的方式——只是必须牢记，这个假设对人类用户才成立。

“基于共识的复制”作为通过用户 ID 进行数据分片的实现细节在系统内部是需要的，因为一个分片的迁移必须被所有客户端精确地、一致地记录下来。如果客户端在失效与存活的副本中来回颠簸，那么会导致类似于邮件忽然消失或重现等用户可见的扭曲。

## 1.4　让系统即时响应

前面两节提出了系统在多台机器、计算中心乃至各大洲分布的论据，用于满足应用的服务范围和可靠性要求。只不过，“重新实现 Gmail”这个练习项目最重要的目标是为终端用户构建一个邮件服务系统，而对于用户来说，无论系统是如何实现的，唯一值得他们在意的指标就是：当他们需要服务时，系统是否能够提供他们所需的服务。换句话说，系统必须快速响应用户发出的任何请求。

完成这个目标的最简单方法当然就是写一个运行在本地环境的应用程序，并且这个应用程序的所有邮件也都保存在本地机器上：跨网络寻求答案永远都会耗费更长时间，而且不如就近保存有答案可靠。也因此在分布式的需求和快速响应的需求之间存在矛盾(tension)。所有分布式设计都必须充分论证，就像 Gmail 例子里面所做的一样。

当分布式结构不可或缺时，你会在探索提高系统响应能力的道路上遇到新挑战。当下很多分布式应用最恼人的行为就是它们的用户交互在网络连接不好时近乎停滞。有意思的是，处理完全没有网络的情形可能比处理缓慢数据流的情形要更简单。在这样的情景下，将要在第 12 章详细讨论的“断路器模式(Circuit Breaker pattern)”就非常有用了。使用这种模式，你可以监控部分功能所需服务的可用性和性能，当服务的质量低于一个阈值(有太多的调用失败或者太高的响应延迟)时就触发断路器，强制将功能切换到不使用该服务的模式下。对于部分组件(服务)不可用的场景，系统需要在最开始设计时就将其纳入考虑范围；而断路器模式则挑明了这种关切。

另一个对即时响应性的威胁来自于应用所依赖的服务可能会出现的瞬时过载。此时待办的请求会积压。即使它们其后会被处理，但是响应延迟仍将比平时更久。这种情况可以通过采用第 16 章描述的“流量控制(flow control)”模式来避免。在 Gmail 例子里面，如下几点需要应用断路器和流量控制模式：

- 运行在用户设备上的前端和提供访问后端功能的 Web 服务器之间；
- Web 服务器和后端服务之间。

第一点的理由前文已经提过了：我们渴求用户可见部分在任何条件下都能保持应用的即时响应，即使它唯一所能做的事情是通知用户服务器宕机了(请求只能在稍后完成)。取决于前端在“离线模式(offline mode)”下能保留多少功能，可能需要停用用户界面的部分区域。

第二点的理由是，如果不这样的话，前端对 Web 服务器的不同请求就可能需要不同的断路器，每个断路器对应一种请求所需要的后端服务的一个子集。如果仅因为后端的一小部分服务不可用就将整个应用切换到离线模式的话，那么这无

疑是无益的过度反应；而如果在前端追踪这些信息，会使得前端的实现耦合于后端的精确结构，导致无论什么时候后端服务的组合改变了，前端的代码也都必须进行相应的调整。因此，Web 服务器这一层应该将相关的细节隐藏掉，并且在所有情况下，都需要给它的客户端尽快提供响应。

拿一个后端服务来举例，这个服务提供展现在联系人卡片上的信息，而这个卡片会在鼠标悬停在邮件发送者姓名上时弹出。以 Gmail 的整体功能考虑，这个模块并非核心组件。所以 Web 服务器可在这个服务不可用时，给所有这类请求返回一个临时不可用的错误码。前端不需要追踪这个状态；它只需要在此时不弹出名片，然后在用户再一次触发了这个交互时重试这个请求即可。

这个推论不仅适用于 Web 服务器这一层。在一个由成百上千个后端服务所组成的大型应用里，以这种方式隔离处理服务失败和不可用同样迫切。否则，系统会因为其行为无法被人类所理解而显得不可理喻。正如功能应该被模块化一样，对于失败条件的处理也必须被封装在一个可以被理解的范围内。

## 1.5 避免大泥球

到目前为止，这个 Gmail 应用已经明确包括：运行在用户设备上的前端部分，提供存储和功能模块的后端服务，以及作为访问后端服务入口的 Web 服务器。后者除了要提供前面讨论的即时响应性之外，还需要服务于另一个重要的目标：在架构上将前端与后端解耦。将客户端请求的入口处明确定义后，有利于简化对于系统组件之间(例如用户设备上的组件和运行在云端服务器的组件之间)的相互作用的推导。

目前后端是由众多服务组成的。这些服务的划分和关系来源于对“简单组件模式”的应用。而在服务内部，这个模式并没有提供关于如何避免架构陷入巨大混乱的制衡机制。在这样的混乱里，每个服务要与几乎所有的其他服务进行通信。这样的系统即使有完美的单独的失败处理机制、断路器和流量控制，也会变得难以管理；它也必定不可能使得开发人员可以完全理解它并有信心做出变更。这种情形被非正式地称为“大泥球(big ball of mud)”。

对于由任意后端服务之间无限制地交互所产生的混乱问题，解决办法是专注于整个应用内的通信链路，并专门设计它们。这样的做法被称作“消息流(message flow)”模式，我们会在第 15 章详细讨论它。

图 1-2 所展示的服务分解粒度过于粗化，不太合适做大泥球的例子，但是可以大概描绘消息流设计的原则，比如说，处理邮件编写的服务不应该直接和处理联系人信息弹出框的服务交互：如果编写一封邮件必须展现邮件里面提及的联系人的名片，那么与其让后端负责做这些事情，不如由前端发出查询弹出信息的请

求，正如前端在用户鼠标悬停在邮件的头信息时所做的事情一样。以这种方式设计，消息流链路所需的数量就可以减少一个，使得后端服务的整体交互模型变得简单一点。

另一个仔细思考消息流设计的好处，会体现在测试的便利上，以及更易于保障交互场景的测试覆盖率。当拥有一个清晰明确的消息流设计后，组件会和哪些服务进行交互、组件所需具备的吞吐量和延迟就会变得显而易见。这个优点也可以被转而当作煤矿里的金丝雀[5]：每当难以为一个给定组件评估哪些场景应该被测试时，那也许就是系统正在成为大泥球的危险信号。

## 1.6　整合非反应式组件

依据反应式原则创建应用的最后一个侧重点在于，应用在大部分情况下都不得不与现有系统或者基础架构进行整合，但这些系统或架构并没有提供反应式应用所需的特性。这样的例子有：缺少封装(在失败时只是简单地终止整个进程)的设备驱动；由于同步执行自身作用(effects)而阻塞调用者，让调用者无法在同一时间对其他输入予以反应或对此次调用触发超时的 API；拥有无界输入队列，但又不遵循“在有限时间内响应”原则的系统。

上述大部分问题都可以使用第 14 章讨论的“资源管理模式(resource-management patterns)”进行处理。基本原理是通过与专用反应式组件内的资源进行交互，按需使用额外的线程、进程或机器，以此改造所需的封装和异步边界。这些资源便可以由此无缝地整合到原有架构中。

当与没有提供有界响应延迟的系统进行交互时，有必要改造出可用信号来通知瞬时过载情形的能力。某种程度上可以通过采用断路器方式来获取这种能力，但是这样的话我们就必须额外考虑对于过载应该给出怎样的响应。第 16 章描述的“流量控制模式(flow-control patterns)”在这种场景下也会有所帮助。

举个在 Gmail 应用背景下的例子，假设有一个与外部功能的整合需求，比如共享的购物清单。在 Gmail 的前端里，用户可通过半自动化手段从邮件抽取出所需的信息，向购物清单里添加物品。后端会通过一个服务封装这个外部 API 来为此功能提供支持。假设与购物清单的交互要求使用原生库，而这个原生库易于崩溃，还能拖垮运行它的进程，那么用专有的进程单独执行这项任务就是值得的。这种外部 API 的封装形式之后就可以通过操作系统的进程间通信(IPC)设施(如管道、套接字或者共享内存)整合进应用。

---

5 英文习语，指某人/某物危险将至的预警标志。煤矿工人过去带着金丝雀下井。这种鸟对危险气体的敏感度超过人。如果金丝雀表现出急躁不安，那么矿工便知道井下有危险气体，需要撤离。——译者注

进一步假设购物清单的实现使用了一个实际上无界的输入队列，这时你就需要思考一下当延迟增加时会发生什么。如果一项物品需要数分钟的时间才能在购物清单里显现，那么用户就会感到困惑甚至烦躁。解决这个问题的其中一个办法是监控购物清单，并且观察它与 Gmail 后端负责与之交互的服务之间的延迟。在当前检测到的延迟超过可接受的阈值时，服务要么在响应时添加拒绝信息，要么回复一个临时不可用的错误码，或者也可以继续执行操作，但是在响应中加入一个警告提示。前端应用之后可以根据任一响应告知用户；要么建议用户稍后重试，要么通知用户可能会有延迟。

## 1.7 小结

在这一章中，我们通过探讨《反应式宣言》所列举的基本原则，大致领略了反应式世界的风景，并通过这种方式来告知你在构建应用时将会面临的主要挑战。如果你需要一个更详细的设计反应式应用的例子，可以参考附录 B。下一章将深入分析《反应式宣言》本身；附录 C 则简述该宣言的要点。

# 附录 C

# 《反应式宣言》正文

版本 2.0，2014 年 9 月 16 日发布。

## C.1 主要内容

在不同领域中深耕的组织都在不约而同地尝试发现相似的软件构建模式。希望这些模式能使系统更健壮、更具回弹性、更灵活，也能更好地满足现代的需求。

近年来，应用程序的需求已经发生了戏剧性变化，模式变化也随之而来。仅在几年前，一个大型应用程序通常拥有数十台服务器、秒级的响应时间、数小时的维护时间以及 GB 级的数据。而今，应用程序被部署到形态各异的载体上，从移动设备到运行着数以千计的多核心处理器的云端集群。用户期望着毫秒级的响应时间，以及服务 100%正常运行(随时可用)。而数据则以 PB 计量。昨日的软件架构已经根本无法满足今天的需求。

我们相信大家都需要一套各类系统都一致合用的架构设计方案，而设计中需要关注的几个维度也已得到各方认可。我们需要系统具备以下特质：即时响应性(Responsive)、回弹性(Resilient)、弹性(Elastic)以及消息驱动(Message Driven)。对于这样的系统，我们称为反应式系统(Reactive System)。

使用反应式方式构建的反应式系统会更灵活、松耦合、可伸缩(参见 C.2.15)。这使得它们的开发过程和调整都更容易。它们对系统的失败(failure)(参见 C.2.7)也更加包容，而当失败确实发生时，它们的应对方案会处理得体而非混乱无序。反应式系统具有高度的即时响应性，为用户(参见 C.2.17)提供了高效的互动反馈。

反应式系统的特质：

- **即时响应性**：只要有可能，系统(参见C.2.16)就会及时做出响应。即时响应是可用性和实用性的基石，而更重要的是，即时响应意味着可快速地检测到问题并有效地对其进行处理。即时响应的系统专注于提供快速而一致的响应时间，确立可靠的反馈上限，以提供一致的服务质量。这种一致的行为转而也将简化错误处理、建立最终用户的信任，并促使用户与系统进一步互动。
- **回弹性**：系统在出现失败(参见 C.2.7)时依然保持即时响应性。这不仅适用于高可用的、任务关键型系统——任何不具备回弹性的系统都将在发生失败后丢失即时响应性。回弹性是通过复制(参见C.2.13)、遏制、隔离(参见 C.2.8)以及委托(参见 C.2.5)来实现的。失败的扩散被遏制在每个组件(参见 C.2.4)内部，与其他组件相互隔离，从而确保系统某部分的失败不会危及整个系统，并能独立恢复。每个组件的恢复都被委托给了另一个外部组件，此外，在必要时也可通过复制来保证高可用性。因此组件的客户端不再承担组件失败的处理。
- **弹性**：系统在不断变化的工作负载之下依然保持即时响应性。反应式系统可对输入(负载)的速率变化做出反应，比如通过增加或者减少被分配用于服务这些输入(负载)的资源(参见 C.2.14)。这意味着在设计上并没有争用点和中央瓶颈，得以进行组件的分片或者复制，并在它们之间分布输入(负载)。通过提供相关的实时性能指标，反应式系统能支持预测式以及反应式的伸缩算法。这些系统可在常规硬件以及软件平台上实现具有成本效益的弹性(参见C.2.6)。
- **消息驱动**：反应式系统依赖异步的(参见C.2.1)消息传递(参见C.2.10)，从而确保了松耦合、隔离、位置透明(参见 C.2.9)的组件之间有着明确边界。这一边界还提供了将失败(参见 C.2.7)作为消息委托出去的手段。使用显式的消息传递，可通过在系统中塑造并监视消息流队列，并在必要时应用回压(参见C.2.2)，从而实现负载管理、弹性以及流量控制。使用位置透明的消息传递作为通信手段，使得跨集群或者在单个主机中使用相同的结构成分和语义来管理失败成为可能。非阻塞的(参见 C.2.11)通信使得接收者可以只在活动时才消耗资源(参见 C.2.14)，从而减少系统开销。

大型系统由多个较小的系统构成，因此整体效用取决于构成部分的反应式属性。这意味着，反应式系统应用一些设计原则，使这些属性能在所有级别的规模上生效，而且可以组合。世界上各类最大型系统所依赖的架构都基于这些属性，而且每天都在服务于数十亿人的需求。现在，是时候在系统设计之初就有意识地

应用这些设计原则了，而不是每次都去重新发现它们。

## C.2 词汇表

### C.2.1 异步

牛津词典把“asynchronous(异步的)”定义为“不同时存在或发生的”。在本宣言的上下文中，我们的意思是：在来自客户端的请求被发送到服务端后，对于该请求的处理可发生这之后的任意时间点。对于发生在服务内部的执行过程，客户端不能直接对其进行观察，或者与之同步。这是同步处理(synchronous processing)的反义词，同步处理意味着客户端只能在服务已经处理完该请求后，才能恢复自己的执行。

### C.2.2 回压

当某个组件(参见 C.2.4)正竭力维持响应能力时，系统(参见 C.2.16)作为一个整体就需要以合理方式作出反应。对于正遭受压力的组件来说，无论是灾难性失败，还是不受控地丢弃消息，都是不可接受的。既然它既不能成功地应对压力，又不能直接失败，它就应该向其上游组件传达其正在遭受压力的事实，并让它们(该组件的上游组件)降低负载。这种回压(back-pressure)是一种重要的反馈机制，使得系统得以优雅地对负载做出反应，而不是在负载下崩溃。回压可一路扩散到(系统的)用户，在这时即时响应性可能有所降低，但这种机制将确保系统在负载之下具有回弹性，并将提供信息，从而允许系统本身通过利用其他资源来帮助分担负载，参见弹性(参见 C.2.6)。

### C.2.3 批量处理

当前计算机为反复执行同一项任务而进行了优化：在 CPU 的时钟频率保持不变的情况下，指令缓存和分支预测增加了每秒可被处理的指令数。这就意味着，快速连续地将不同的任务递交给相同的 CPU 核心，将并不能获益于本有可能得到的完全(最高利用率的)性能：如有可能，我们应该这样构造应用程序，它的执行逻辑在不同任务之间交替的频率更低。这就意味着可成批处理一组数据元素，这也可能意味着可在专门的硬件线程(指 CPU 的逻辑核心)上执行不同处理步骤。

同样的道理也适用于对于需要同步和协调的外部资源(参见 C.2.14)的使用。当从单一线程(即 CPU 核心)发送指令，而不是从所有的 CPU 核心争夺带宽时，由

持久化存储设备所提供的 I/O 带宽将得到显著提高。使用单一入口的额外效益，即多个操作可被重新排序，从而更好地适应设备的最佳访问模式(当今的存储设备的线性存取性能要优于随机存取的性能)。

此外，批量处理还提供了分摊昂贵操作(如 I/O)或者昂贵计算的成本的机会。例如，将多个数据项打包到同一个网络数据包或者磁盘存储块中，从而提高效能并降低使用率。

### C.2.4 组件

我们所描述的是一个模块化的软件架构，它实际上是一个非常古老的概念，参见 Parnas(1972)。我们使用“组件(component)”这个术语，因为它和“隔间(compartment)”联系紧密，其意味着每个组件都是自包含的、封闭的并和其他组件隔离。这个概念首先适用于系统的运行时特征，但它通常也会反映在源代码的模块化结构中。虽然不同的组件可能使用相同的软件模块来执行通用的任务，但定义了每个组件的顶层行为的程序代码则是组件本身的一个模块。组件边界通常与问题域中的限界上下文(Bounded Context)紧密对齐。这意味着，系统设计倾向于反应问题域，并因此在保持隔离的同时也更容易演化。消息协议(参见 C.2.12)为多个限界上下文(组件)之间提供了自然的映射和通信层。

### C.2.5 委托

将任务异步地(参见 C.2.1)委托给另一个组件(参见 C.2.4)意味着该任务将在另一个组件的上下文中执行，举几个可能的情况：这个被委托的内容甚至可能意味着运行在不同的错误处理上下文里，属于不同的线程，来自不同的进程，甚至在不同的网络节点上。委托的目的是将处理某个任务的职责移交给另一个组件，以便发起委托的组件可执行其他处理，或者有选择地观察被委托的任务的进度，以防需要执行额外的操作(如处理失败或者报告进度)。

### C.2.6 弹性(与“可伸缩性”相对)

弹性意味着当资源根据需求按比例地减少或者增加时，系统的吞吐量将自动向下或者向上缩放，从而满足不同需求。系统需要具有可伸缩性(参见 C.2.15)，以使其可从运行时对资源的动态添加或者删除中获益。因此，弹性是建立在可伸缩性的基础之上的，并通过添加自动的资源(参见 C.2.14)管理概念对其进行了扩充。

### C.2.7 失败(和“错误”相对)

失败是一种服务内部的意外事件，会阻止服务继续正常运行。失败通常会阻止对当前客户端请求(并可能是接下来的所有客户端请求)的响应。与错误相对照，错误是意料之中的，并针对各种情况进行处理(例如，在输入验证的过程中所发现的错误)，将作为该消息的正常处理过程的一部分返回给客户端。而失败则是意料之外的，并在系统(参见 C.2.16)能恢复至和之前相同的服务水平之前，需要进行干预。这并不意味着失败总是致命的(fatal)，虽然在失败发生之后，系统的某些服务能力可能会降低。错误是正常操作流程预期的一部分，在错误发生后，系统将立即对其进行处理，并将继续以相同的服务能力运行。

失败的例子有：硬件故障、由于致命的资源耗尽而引起的进程意外终止，以及导致系统内部状态损坏的程序缺陷。

### C.2.8 隔离(和“遏制”相对)

隔离可定义为在时间和空间上的解耦。在时间上解耦意味着发送者和接收者可拥有独立的生命周期——它们不需要同时存在，从而使得相互通信成为可能。通过在组件(参见 C.2.4)之间添加异步(参见 C.2.1)边界，以及通过消息传递(参见 C.2.10)实现了这一点。在空间上解耦(定义为位置透明性(参见 C.2.9))意味着发送者和接收者不必运行在同一个进程中。不管运维部门或者运行时本身决定的部署结构是多么高效——在应用程序的生命周期之内，这一切都可能会发生改变。

真正的隔离超出了大多数面向对象的编程语言中所常见的封装概念，并使得我们可以对下述内容进行划分和遏制：

**状态和行为**：它支持无共享的设计，并最大限度地减少了竞争和一致性成本(如通用伸缩性原则(Universal Scalability Law)中所定义的)；

**失败**：它支持在细粒度上捕获、发出失败信号以及管理失败(参见 C.2.1)，而不是将其扩散(cascade)到其他组件。

组件之间的强隔离性是建立在基于明确定义的协议(参见 C.2.12)的通信之上的，并支持解耦，从而使得系统更加容易被理解、扩展、测试和演化。

### C.2.9 位置透明性

弹性系统(参见 C.2.6)需要能够自适应，并不间断地对需求的变化做出反应。它们需要优雅而高效地扩大或者缩减(部署)规模。极大地简化这个问题的一个关键洞察是：认识到我们一直都在处理分布式计算。无论我们是在单个(具有多个独立 CPU，并通过快速通道互联(QPI)通信的)节点之上，还是在一个(具有多台通

过网络进行通信的独立节点的)机器集群之上运行系统，都是如此。拥抱这一事实意味着，在多核心之上进行垂直缩放和在集群之上进行水平伸缩并没有什么概念上的差异。

如果所有组件(参见 C.2.4)都支持移动性，而本地通信只是一项优化。那么我们根本不需要预先定义一个静态的系统拓扑和部署结构。可将这个决策留给运维人员或运行时，让他(它)们根据系统的使用情况来对其进行调整和优化。

这种通过异步的(参见 C.2.1)消息传递(参见 C.2.10)实现的在空间上的(请参见隔离的定义，C.2.8)解耦，以及将运行时实例和它们的引用解耦，就是我们所谓的位置透明性。位置透明性通常被误认为是“透明的分布式计算”，然而实际上恰恰相反：我们拥抱网络，以及它所有的约束——如部分失败、网络分区、消息丢失，以及它的异步性和与生俱来的基于消息的性质，并将它们作为编程模型中的一等公民，而不是尝试在网络上模拟进程内的方法调用(如 RPC、XA 等)。我们对于位置透明性的观点与 Waldo 等人所著的 *A Note On Distributed Computing* 中的观点完全一致。

## C.2.10 消息驱动(与“事件驱动”相对)

消息是指发送到特定目的地的一组特定数据，事件是组件(参见 C.2.4)在达到了某个给定状态时发出的信号。在消息驱动的系统中，可寻址的接收者等待消息的到来，并对消息做出反应，否则只是休眠(即异步非阻塞地等待消息的到来)。而在事件驱动的系统中，通知监听器被附加到事件源，以便在事件被发出时调用它们(指回调)。这也意味着，事件驱动的系统关注可寻址的事件源，而消息驱动的系统则着重于可寻址的接收者。消息可包含编码为有效载荷的事件。

由于事件消耗链的短暂性，所以在事件驱动的系统中很难实现回弹性：当处理过程已经就绪，监听器已经设置好，以便响应结果并对结果进行变换时，这些监听器通常都将直接处理成功或失败(参见 C.2.7)，并向原始的客户端报告执行结果。这些监听器响应组件的失败，以便恢复它(指失败的组件)的正常功能，另一方面，需要处理的是那些并没有与短暂的客户端请求捆绑在一起，但影响整个组件的健康状况的失败。

## C.2.11 非阻塞的

在并发编程中，如果保护资源的互斥并没有把争夺资源的线程无限期地推迟执行，那么该算法则被认为是非阻塞的。在实践中，这通常缩减为一个 API，当资源可用时，该 API 将允许访问该资源(参见 C.2.14)，否则它将立即返回，并通知调用者该资源当前不可用，或者该操作已经启动了，但尚未完成。某个资源的

非阻塞 API 使得其调用者可执行其他操作，而不是被阻塞以等待该资源变为可用。此外，还可允许资源的客户端注册，以便让其在资源可用时，或者操作已经完成时获得通知。

### C.2.12 协议

协议定义了在组件(参见 C.2.4)之间交换或者传输消息的方法与规范。协议由会话参与者之间的关系、协议的累计状态以及允许发送的消息集所构成。这意味着，协议描述了会话参与者在何时可发送何种消息给另一个会话参与者。协议可按其消息交换的形式进行分类，一些常见的类型是：请求-响应模式、重复的请求-响应模式(如 HTTP 中)、发布-订阅模式以及(反应式)流模式。

与本地编程接口相比，协议则更通用，它可包含两个以上的参与者，并可预见到消息交换的进展，而接口仅指定调用者和接收者之间每次一个交互的过程。

需要注意，这里定义的协议只指定可能发送什么消息，而不是它们应该如何被编码、解码，对于使用该协议的组件来说，传输机制是透明的。

### C.2.13 复制

在不同的地方同时执行一个组件(参见 C.2.4)被称为复制。这可能意味着在不同的线程或线程池、进程、网络节点或计算中心中执行。复制提供了可伸缩性(参见 C.2.15)(传入的工作负载将会被分布到跨组件的多个实例中) 以及回弹性(传入的工作负载将被复制到多个能并行处理相同请求的多个实例中)。这些方式可结合使用，例如，在确保该组件的某个确定用户的所有相关事务都将由两个实例执行的同时，实例的总数又根据传入的负载而变化(参见弹性，C.2.6 节)。

在复制有状态的组件时，必须小心地同步副本之间的状态数据，否则该组件的客户需要知道同步的模式，并且违反了封装的目的。通常，同步方案的选择需要在一致性和可用性之间进行权衡，如果允许被复制的副本在有限的时间段内不一致(最终一致性)，将得到最佳的可用性，同时，完美的一致性要求所有复制副本以步调一致(lock-step)的方式来推进它们的状态。在这两种“极端”之间存在着一系列可能的解决方案，所以每个组件都应该选择最适合自己需要的方式。

### C.2.14 资源

组件(参见 C.2.4)执行其功能所依赖的一切都是资源，资源必须根据组件的需要而进行调配。这包括 CPU 的分配、内存、持久化存储以及网络带宽、内存带宽、CPU 缓存、内部插座间的 CPU 链接、可靠的计时器以及任务调度服务、其

他输入和输出设备、外部服务(如数据库或者网络文件系统等)等。所有这些资源都必须考虑弹性(参见 C.2.6)和回弹性，因为缺少必需的资源将妨碍组件在需要时发挥正常作用。

## C.2.15 可伸缩性

一个系统(参见 C.2.16)通过利用更多计算资源(参见 C.2.14)来提升其性能的能力，是通过系统吞吐量的提升与资源增加量的比值来衡量的。一个完美的可伸缩性系统的特点是这两个数字是成正比的。所分配的资源翻倍也将使得吞吐量翻倍。可伸缩性通常受限于系统中所引入的瓶颈或者同步点，参见阿姆达尔定律以及 Gunther 的通用可伸缩模型(Amdahl's Law and Gunther's Universal Scalability Model)。

## C.2.16 系统

系统为它的用户(参见 C.2.17)或客户端提供服务。系统可大可小，它们可包含许多组件或只有少数几个组件(参见 C.2.4)。系统中的所有组件相互协作，从而提供这些服务。很多情况下，位于相同系统中的多个组件之间，具有某种客户端-服务端的对应关系(例如，考虑一下，前端组件依赖于后端组件)。一个系统中共享一种通用的回弹性模型，意即，某个组件的失败(参见 C.2.7)将在该系统的内部得到处理，并由一个组件委托(参见 C.2.5)给另一个组件。如果系统中的某系列组件的功能、资源(参见 C.2.14)或者失败模型都和系统中的其余部分相互隔离，将这一系列组件看成是系统的子系统将更有利于系统设计。

## C.2.17 用户

我们使用这个术语来非正式地指代某个服务的任何消费者，可以是人类或者其他服务。