Broadview
www.broadview.com.cn

Pearson

响应式架构

消息模式Actor实现与Scala、Akka应用集成

[美] **Vaughn Vernon** 著
苏宝龙 译

Akka项目创始人Jonas Bonér为本书作序

Reactive Messaging Patterns with the Actor Model

Applications and Integration in Scala and Akka

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

响应式架构

消息模式Actor实现与Scala、Akka应用集成

[美] Vaughn Vernon 著
苏宝龙 译

Reactive Messaging Patterns

with the

Actor Model

Applications and Integration in Scala and Akka

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

内容简介

通过Actor模型使用响应式消息传输模式，可编写出具有高性能、高响应性、高可伸缩性和高韧性的并发应用程序。本书由10章构成，详细介绍了使用Actor模型中的响应式消息传输模式的理论和实用技巧。其中包括：Actor模型和响应式软件的主要概念、Scala语言的基础知识、Akka框架与Akka集群功能、Actor模型中的通道机制和技术、降低消息源与消息目的地之间耦合性的方式、持久化Actor对象和幂等接收者。附录A中还介绍了通过.NET平台和C#语言使用Actor模型的方式。

在企业中任职的软件架构师和开发者，以及任何对Actor模型感兴趣并渴望提高自身技术和价值的软件开发者，均适合阅读本书。

版权贸易合同登记号 图字：01-2015-7885

图书在版编目（CIP）数据

响应式架构：消息模式Actor实现与Scala、Akka应用集成 /（美）沃恩·弗农（Vaughn Vernon）著；苏宝龙译.—北京：电子工业出版社，2016.7
书名原文：Reactive Messaging Patterns with the Actor Model: Applications and Integration in Scala and Akka
ISBN 978-7-121-29113-5

Ⅰ.①响… Ⅱ.①沃… ②苏… Ⅲ.①程序语言-程序设计 Ⅳ.①TP312

中国版本图书馆CIP数据核字（2016）第137490号

策划编辑：张春雨
责任编辑：刘　舫
印　　刷：北京中新伟业印刷有限公司
装　　订：北京中新伟业印刷有限公司
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：27.5　　字数：555千字
版　　次：2016年7月第1版
印　　次：2016年7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

终于有一本围绕企业应用和架构来讲解 Actor 模型和 Akka 的书了。很期待这类书的出现，希望能引领 Actor 模型开始向企业应用的“回归”。

本书作者 Vaughn Vernon 恰好也是《实现领域驱动设计》（*Implementing Domain Driven Design*）一书的作者，这在某种程度上印证了，近十来年的“领域驱动设计（DDD）”理想在 Actor 模型和 Akka 上终于找到了现实的技术实现。

DDD 希望能在业务领域层面就把模型和逻辑设计清楚（业务模型和逻辑是最稳定的），并一一对应到实现中。或者说，领域有什么，实现中才应该有，也应该有。但由于计算机性能限制、语言实现难度等方面的原因，这一理想在现实中一直没能得到很大程度的实现。

而近二十年来，Java 及其生态一直占据着企业应用领域的主导地位。尤其从 20 世纪 90 年代末以来，J2EE 规范（现在叫 Java EE）也试图围绕业务领域，为企业应用提供从建模到分层，涵盖事务、持久化、分布式的整套解决方案，并提出 Entity Bean、Session Bean 及 Message Bean 等试图对应到业务领域的模型。

遗憾的是，基于当时的技术，Java EE 也并没有很好地实现初衷，这在我看来至少有以下几方面的原因：

- EJB 规范起初的一个主要价值——对分布式应用进行事务管理，在实践中几乎很少被使用，反倒引入了整个架构的复杂性。
- Entity Bean 不能在集群中分片部署，而这本应是分布式系统最需要解决的问题之一。
- 只有 Message Bean 是异步的，但它却不是 Entity Bean。这意味着系统很难在时间维度解耦。

2005 年前后，Hibernate、Spring 等技术逐渐兴起，以轻量化的角度切入了企业应用领域，并在互联网领域异军突起。

对于 Spring + Hibernate 的方案，保存在持久层的业务实体的数据 / 状态

需要反复被业务逻辑存取。为了解决这个性能瓶颈，不得不引入 Ehcache、Memcached、Redis 等中间缓存。这样，在数据扩张时，为了解决这些缓存数据的分片（sharding）问题，这些缓存方案还需要进一步引入和实现集群分片的支持，这带来了复杂性。可即便如此，它解决的是缓存数据的分布，而并没有解决 Beans 本身的分布。Beans 仍然受限于 Bean 容器的缓存大小，而不得不经常去中间缓存甚至持久层要数据。

那么，在企业应用领域，Actor 模型能带来更合适的解决范式吗？

Carl Hewitt 在 1973 年对 Actor 模型进行了如下定义："Actor 模型是一个把'Actor'作为'并发数字计算的通用原语'的数学理论。"。这个定义跟我常说的"Actor 是最适合并行计算的最小颗粒"是相通的。

Actor 是异步驱动、可以并行和分布式部署及运行的最小颗粒。也即，它可以被分配、分布、调度到不同的 CPU、不同的节点，乃至不同的时间片上运行，而不影响最终的结果。因此，Actor 是在空间（分布式）和时间（异步驱动）上解耦的。

Akka 是 Lightbend（前身是 Typesafe）公司在 JVM 上的 Actor 模型实现，它同时是一个可扩展、引入了多种分布式范式的框架。而且，Akka 2.3.0 开始支持带状态的 Actors 的分片集群，以及根据 journal/snapshot 形式对事件流和状态快照实施持久化和回放。

Akka 的 Actor 模式本身可以保证在单个 Actor 实例中每个行为的原子性，并行的粒度可以细化到单个 Actor 实例。也即，当行为被封装在一个 Actor 实例中时，该行为不会阻塞其他 Actors 实例的行为，从而很难出现整个系统被阻塞的情况。

从 EJB 角度看，Akka Actor 提供了什么对应的角色呢？

首先，从各类 Bean 的角色看。Akka Actor 支持持久化，所以有一类 Actor 可以设成"Entity Bean"；Actor 实例可以维护自己的状态，所以它也可以是"Stateful Session Bean"；而不需要关心其状态的 Actor，自然可以担当"Stateless Session Bean"；最后，对 Actor 的存取、调用，都是通过异步的消息传递来实现的，因此，它们都是"Message Bean"。

其次，从架构层面看。Actor 能同时担当实体 Beans 和中间缓存的角色，并且是异步驱动的，且具备分片集群下的水平扩展能力。而 akka-persistent 进一步将持久化、HA 一并细粒度地实现了。与 SSH/Java EE 相比，Actor 减少了数据反复在各种形态（数据库、缓存、业务层中的实体对象实例）间转化的消耗，减少了线程阻塞的消耗，并提供了一致的并行和分布式机制。

再次，从业务领域看。Actor 可以非常自然地直接对应到业务实体。某类 Actor 的一个实例可以是一个人、一个物品、一部设备，等等。这些实体 Actor 都

是通过接收命令或者事件，来驱动完成一次状态的变化或者完成一次任务会话。Akka 的每个 Actor 仍由自己的 scheduler 来完成各类定时任务，也可以理解为它们同时可以由时钟事件来驱动。而一次任务会话也可以抽象为一个会话 Actor 的实例，它跟踪会话的进度、事务状态、发送事件等。

最后，就像前面提到的，对 Actor 实例的存取、调用是通过异步的消息传递来实现的，这带来一种担心：性能代价会不会很高？

这确实是很多年前，采用异步消息驱动来设计、实现系统的架构师、程序员的最大障碍，因为那时的机器不仅性能有限，多核的物理机制也少见且昂贵，而且多线程的切换代价高昂。

但对于现在的计算机而言，多核处处可见，CPU 的计算能力、线程的调度和切换能力也有了极大的提高，上面这些问题已经不再是障碍。比如，Akka 在单核的 CPU 上，每秒可以处理的异步消息数是 5000 万以上。尤其是，现代 JVM 的线程切换时间已经在微秒级，线程切换的代价变得非常小。小到在大量场景下，采用异步处理所带来的整体性能和效率的提升已经足够将其代价忽略。

还有一种担心来自：用 Akka 会引入复杂的架构吗？

从 DDD 的理念和实践来看，恰恰相反，因为处处都是同样的模型（Actor、Async Message Driven、Event Streaming），系统实际上更具一致性和弹性（包括对需求的弹性）。

这些年，我本人一直在尝试将 Actor 模式及 Akka 用于企业应用，包括证券交易及计算领域，以及在豌豆荚实现的若干实时系统。实际上最近十年做来做去无非是同一件事：将现实（直接）映射到计算机体系的 Actor 模型。而实践效果充分验证了，采用异步消息驱动以及小粒度的并行和持久化机制，在性能和内存的使用上都不是问题。更重要的是，模型及架构与领域的自然对应大大降低了系统进化和维护的成本。而得益于 Akka 的集群、高可用及事件溯源（Event-Souring）的持久化机制，这几个系统也几乎都能无故障地持续运行。运行时间最长的一个，超过了两年没有重启。

本书的内容是基于 Akka 2.3.4 版本的，这个版本包含了 Akka 框架主要的功能和实现（包括 sharding 和 persistent），非常新且全面。而且作为一个长期从事企业应用领域的设计和实现的专家，作者非常熟悉在企业业务领域需要用到的知识和术语以及思维方式，并很好地融入了 Akka 的实践。

——豌豆荚资深程序员　邓草原

译者序

1965 年 Intel 的创始人戈登 · 摩尔发现了摩尔定律，50 多年来，计算机的性能一直遵循摩尔定律迅猛发展：CPU 可容纳的晶体管数目，每隔约 18 个月便会增加一倍，性能也将提升一倍。如今 CPU 中晶体管的数量以指数形式增长的迅猛势头似乎要走到尽头了。而计算机性能的另一要素——CPU 主频速度的提高，早在 2003 年就开始急剧下降。计算机的性能无法迅猛增长的同时，人们的需求却仍旧以指数形式增长，供求矛盾日益尖锐。

传统提高 CPU 性能的技术已经被多核和超线程技术取代。事实证明，硬件工程师再也不能独自承担提高计算机性能的重任了。当前硬件工程师确实能够设计出含有 288 个核心的 CPU，但如果该 CPU 没有被用于运行相应的并发程序，这个含有 288 个核心的 CPU 只能被当作单核 CPU 使用。

该是软件工程师挺身而出勇挑重担的时候了。但是，使用传统的并发技术（如线程、锁和监控器等）开发软件会遇到许多难以克服的难题。例如，到软件开发过程的末尾阶段，客户提出增加功能的要求，或者需要对某个（些）功能进行改进，就不得不重新调整线程的分配和几乎所有并发分支。这些工作量可能不比重新开发一个新的软件少多少，甚至可能会比开发新的软件更加困难。因此，传统并发技术注定不能担任当前软件开发工作的主角。开发者们迫切需要的是高级并发编程技术。

Carl Hewitt 博士早在 20 世纪 70 年代初就发明了 Actor 模型，这种优秀的高级并发编程思想超越了 Carl Hewitt 博士所处的时代。但当时功能最强大的处理器也无法将该理论付诸实践。直到多核处理器、云计算、移动设备和互联网无处不在的今天，Actor 模型才重新焕发了青春。

Actor 模型拥有下列优点：

1. 大幅度降低应用程序内部的耦合性。
2. Actor 模型的消息传递形式简化了并行程序的开发工作，使开发人员无须

与并发编程基础元素打交道。

3. 在高动态环境中，Actor 模型既可以利用顺序编程技巧，也可以利用函数编程技巧。
4. Actor 模型可以解决许多并发编程难题，如死锁、活锁、互斥体等。
5. Actor 模型能够大幅度提高调用方法的安全性和速度。

凭借 Actor 模型的这些优势，通过响应式消息传输模式，开发者能够开发出具有高性能、高响应性、高可伸缩性和高韧性的并发应用程序。

本书的作者 Vaughn Vernon 是一位资深的软件开发者，并且是一位简化软件设计和实现思想的领袖人物。他在本书中使用了大量的实践案例，这些范例程序既有实用性也有启迪性，深入浅出地讲解了使用 Actor 模型通过 Scala 语言和 Akka 框架，编写响应式应用程序的理论和实用技巧。

翻译前沿计算机科学书籍的工作并不轻松，也不是单独一个人能够完成的。在此我要感谢电子工业出版社计算机出版分社的张春雨等编辑对本书提供的帮助。此外，石浩、孙顾、徐颖、朱晶晶、沈骏杰、何志颖、许诗怡、马佳妮、尹晓婷、徐雯、郭昕、陆迎明和孙艳婷等也参与了本书的翻译工作。

因时间仓促，译者水平有限，本书的错漏之处欢迎广大读者朋友们批评指正。

序

20 世纪 70 年代初，Carl Hewitt 发明了 Actor 模型，他超越了自己所处的时代。他通过 Actor 概念，定义了一个含有不确定性的计算模型（假设所有计算操作都是通过异步方式执行的）。该模型使用并发处理模式和有稳定的状态独立处理过程的概念，全方位地降低了 Actor 对象的耦合性，并使之支持分布式和移动架构。

当前，软件行业已经跟上了 Carl Hewitt 的创新思路；多核处理器、云计算、移动设备和互联网都成为常见的事物。这从根本上改变了软件行业，而且使创建并发模型和分布式处理基础理论的需求变得更为迫切。我相信 Actor 模型能够成为我们迫切需要的坚实理论基础，使我们能够通过具有响应性、韧性和弹性的响应式编程原则，创建复杂的分布式系统以应对当前的挑战。这就是我编写 Akka 框架的原因：将 Actor 模型的强大功能交到普通开发者手中。

看到 Vaughn Vernon 撰写的这本书我感到非常兴奋。这本书介绍了大家都很需要了解的知识——将 Actor 对象与传统的企业级消息传输系统连接起来，以及使用 Actor 对象创建响应式应用程序的方式。我喜欢这本书仅依赖 Akka 框架基础功能（是 Actor 模型而不是 Akka 框架中的高级库）来介绍高级消息传输和通信模式的方式。即使 Actor 模型仅是一种低等级的计算模型，但看到使用它可以通过简单直观的方式实现功能强大且多样的消息传输模式，确实是一件令人赏心悦目的事情。一旦你了解了基础的编程思路，就能够向其中添加高级工具和技巧。

这本书还介绍了许多形式化和命名模式，这是 Akka 社区成员通过数年的研究探索和反复改进获得的成果。这使我回忆起几年前，读到 Gregor Hohpe 和 Bobby Woolf 撰写的经典著作 *Enterprise Integration Patterns*[EIP] 时的惊喜之情。我对 Vaughn Vernon 能够继承这本经典著作的精髓，并对其做了全新的诠释感到很高兴。但我认为这本书最重大的贡献在于，它并没有止步于前人探索过的区域，而是为 Actor 对象的消息传输操作定义了一种独特的模式化语言。这使我们能够

使用专业术语来思考、讨论和交流 Actor 对象传输消息的模式和编程思路。

不论你是初学者还是资深的编程高手，这本书都能够为你提供重要帮助。我希望你能够和我一样与它成为好朋友。

Jonas Bonér

Akka 项目的创始人

前言

当前，许多软件项目折戟沉沙。各种调查和研究报告表明，软件项目的失败比例为 30% 至 50%。该统计数字还不包括那些已经被交付使用，但没有达到某些成功标准的软件项目。当然，该统计数字中包含了企业级的软件项目。上述数据摘自 *Dr. Dobb's Journal*[DDJ] 杂志中由 Scott Ambler[Ambysoft] 撰写的调研报告 *Chaos Report*[Chaos Report]。

与此同时，许多公司使用 Scala 语言和 Akka 框架在突破性能限制和获得可伸缩性方面，取得了令人瞩目的成功 [WhitePages]。这些成功不仅代表软件项目的成功，还代表了在非功能性需求方面取得的成功。当然，这些成功不是仅通过 Scala 语言和 Akka 框架获取的，但同时也不应抹杀 Scala 语言和 Akka 框架在这些成功中所起的重要作用。我确信开发者们是根据他们选择的平台使用这些工具的，这也是他们获得成功的关键。

几年来，我一直希望将 Scala 语言和 Akka 框架引荐给数量众多的想要获取上述成功的企业。本书的目标是使你熟悉 Actor 模型并通过 Scala 语言和 Akka 框架实现 Actor 模型的方式。此外，我相信许多企业级软件架构师和开发者已经通过 Gregor Hohpe 和 Bobby Woolf 撰写的经典著作 *Enterprise Integration Patterns*[EIP] 获得了启迪。*Enterprise Integration Patterns* 中介绍了 65 种整合模式，这些模式能够帮助开发团队将企业中的各种独立系统整合到一起。我认为通过 Actor 模型来使用这些整合模式，除了能够使这些模式非常得心应手外，还会使架构师和开发者获得如虎添翼的感觉。

当通过 Actor 模型使用这些整合模式时，与其他方式的主要差异是使用这些整合模式的原始动机。在通过 Actor 模型使用这些整合模式时，这些整合模式被用于开发新的应用程序，而不仅仅是被用于整合旧的应用程序。这是因为这些模式首先是消息传输模式，而后才是整合模式，而 Actor 模型的本质就是消息传输。当使用领域驱动设计 [DDD,IDDD] 方式时，你还会发现一些比较高级的整合模式

（如处理过程管理器），可以帮助你通过直观的方式为突出的业务概念创建模型。

本书面向的读者

本书面向在企业中任职的软件架构师和开发者，以及任何对 Actor 模型感兴趣并渴望提高自身技术和价值的软件开发者。尽管本书着重介绍的是 Scala 语言和 Akka 框架，但在附录 A 中介绍了通过 .NET 平台和 C# 语言使用 Actor 模型的方式。

本书的主要内容

第 1 章介绍 Actor 模型和响应式软件的主要概念。第 2 章介绍 Scala 语言的基础知识，以及 Akka 框架与 Akka 集群功能。第 3 章介绍计算机性能的发展史和未来趋势，以及凭借 Scala 语言和 Akka 框架的可伸缩性，在企业级软件中通过 Actor 模型获得高性能和高可伸缩性的来龙去脉。

之后，本书共使用 7 章内容介绍 Actor 模型中的消息传输模式。第 4 章介绍基础消息传输模式，并概要介绍了后面 5 章的主要内容。第 5 章介绍基础的通道机制和几种通道技术，在使用这些通道技术应对各种应用程序开发和整合挑战时，每种通道技术都能够发挥本身独特的优势。第 6 章介绍消息的结构，指明消息中必须包含消息发送者与消息接收者进行通信的原因。第 7 章介绍降低消息源与消息目的地之间耦合性的方式，以及在消息路由器中放置合适的业务逻辑的方式。第 8 章详细介绍在应用程序开发和整合环境中，需要使用的各种消息转换方式。第 9 章介绍各种各样的消息端点，其中包括持久化 Actor 对象和幂等接收者。最后，第 10 章介绍更高级的编程模式、基础结构和调试工具。

本书的约定

本书主要介绍各种消息传输模式。你不必尝试一鼓作气地阅读所有这些模式，但通常你应该了解第 4 章至第 10 章的内容，以便知道各种模式的详细介绍在书中的具体位置。这样，当你需要掌握某种消息传输模式时，就能够迅速找到介绍它的内容。本书为每种消息传输模式都提供了一个标志性的配图，而且每种模式通常都会拥有插图和源代码（至少会有一个插图和一段代码），以详细介绍使用 Scala 语言和 Akka 框架实现该模式的方式。

本书介绍的各种消息传输模式，实际上一起构成了一种模式语言，这种模式语言由各种相互关联的开发模式和公式（如将基于内容的路由器与死信通道路由器配合使用）构成。在设计基于消息的应用程序和系统时，就可以使用这种模式语言。因此，了解各种模式之间的依赖关系十分必要，正是这些模式之间相互支持的关系构成了这门完整的语言。因此，本书提到某种模式时，会使用*模式名称*表示。

本书的第二个约定是使用插图表示消息传输模式和 Actor 模型的方式。我与 Typesafe 公司的 Roland Kuhn 和 Jamie Allen 一起制定了这些约定。这两个伙伴和我一起撰写了与本书题材类似的新书 *Reactive Design Patterns*，这本书很快会与大家见面。我想使用相同风格的插图介绍这两本书中的 Actor 模型，因此我找 Roland Kuhn 和 Jamie Allen 一起进行了讨论。下面是我们的讨论结果。

如图 P.1 所示，Actor 对象由圆圈代表，而且通常会将该对象的名称放在圆圈中。使用这种表示的多种原因之一是，Gul Agha 很久以前在他撰写的 *Actors: A Model of Concurrent Computation in Distributed Systems*[Agha, Gul] 一书中就使用了这种表示方式。

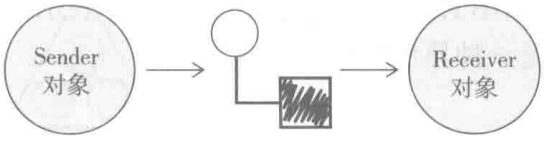

图 P.1　Actor 对象 Sender 向 Actor 对象 Receiver 发送了一条消息。

此外，*Enterprise Integration Patterns*[EIP] 一书也使用这种表示方式将消息表示为组件，因此我们延续了这种用法。带箭头的线条指明了传输消息的源头和目的地。通过图 P.2 和图 P.3 展示的表示方式，你可以将持久化 Actor 对象（长期存在的）和临时 Actor 对象（短期存在的）区分开。代表持久化 Actor 对象的圆圈是由实线绘制的。持久化是指该 Actor 对象会长期存在。持久化并不单指将 Actor 对象存储在硬盘中，也涵盖将 Actor 对象存储在其他永久性存储设备中。另一方面，代表临时 Actor 对象的圆圈是由虚线绘制的。临时是指该 Actor 对象仅会短期存在，在执行完特定任务后会被停止。

图 P.2　持久化 Actor 对象（长期存在的）。

图 P.3 临时 Actor 对象（短期存在的）。

如图 P.4 所示，一个 Actor 对象可以创建另一个 Actor 对象，这就形成了一种父子关系。一个小圆圈外套一个大圆圈和带箭头的线条代表了这种创建操作。其中带箭头的线条与表示消息传输的方式相同，这是因为创建子 Actor 对象的处理过程是一种异步操作。

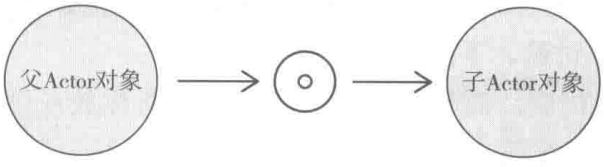

图 P.4 父 Actor 对象创建了一个子 Actor 对象。

Actor 对象的自行终止操作是通过向自己发送特殊的消息实现的。该特殊消息由外套圆圈的 X 和带箭头的线条代表，如图 P.5 所示。该操作也使用了表示消息传输的方式，因为这种自行终止操作也是一种异步操作。

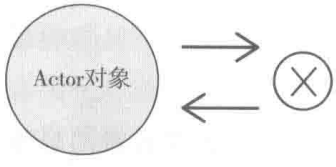

图 P.5 Actor 对象的自行终止操作。

一个 Actor 对象停止另一个 Actor 对象的操作也是通过相同的特殊消息实现的，但该消息是由一个 Actor 对象发送给另一 Actor 对象的。图 P.6 展示了父 Actor 对象停止它的子 Actor 对象的方式。

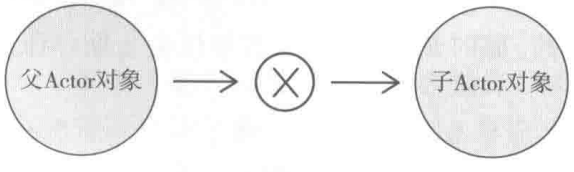

图 P.6 一个 Actor 对象停止另一个 Actor 对象。

通过统一建模语言（UML）顺序图可以表现 Actor 对象的生命周期，如图 P.7 所示。Actor 对象在其生命周期中收到的消息由小圆圈代表（与大头针类似）。你

必须认识到每条消息都是以异步方式接收的。

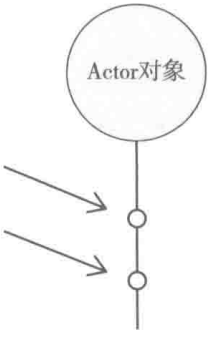

图 P.7　该 Actor 对象收到的两条消息展示了该 Actor 对象的生命轨迹。

父子层次结构由代表父 Actor 对象的圆圈和代表子 Actor 对象集合的三角形构成，如图 P.8 所示。

图 P.8　Actor 对象的父子层次结构。

一个 Actor 对象可以通过赋值或介绍操作认识另一个 Actor 对象。赋值操作是指在创建一个 Actor 对象时，将该 Actor 对象的引用赋予另一个 Actor 对象。另一方面，通过发送消息可以将一个 Actor 对象介绍给另一个 Actor 对象。

如图 P.9 所示，介绍操作由带箭头的虚线代表，该虚线会由被介绍的 Actor 对象指向被发送给另一个 Actor 对象的消息。在本例中，由父 Actor 对象创建的子 Actor 对象被介绍给了消息的接收者。

最后如图 P.10 所示，消息的发送次序由顺序编号代表。该图中含有两条拥有编号 2（代表步骤 2）的消息，和两条拥有编号 3（代表步骤 3）的消息并非错误。这代表一种并发处理可能性，这两条消息可能被同时发送。在本例中，路由器设置了一个计时器（如图 P.10 的上方所示），并通过并发方式向接收者发送了一条消息（步骤 2），同时向计时器发送了一条开始计时的消息（步骤 2）。路由器可能无法在计时器设置的时限内收到接收者的回复消息，此时计时器会向路由器发送一条超时消息（步骤 3）。如果路由器在计时器设置的时限内收到了接收者的回

复消息（步骤 3），那么客户端就会在步骤 4 中收到积极的确认消息。否则，客户端就会在步骤 4 中收到表明该操作超时的消息。

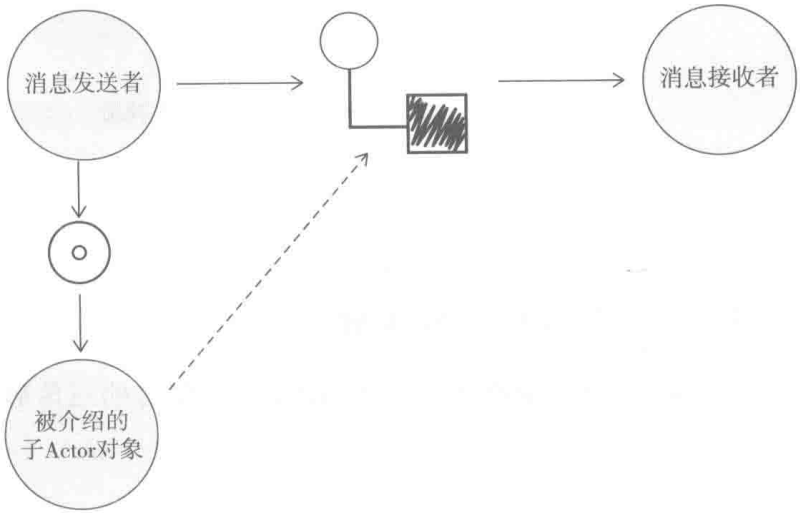

图 P.9　消息发送者通过向消息接收者发送消息，将它的子对象介绍给消息接收者。

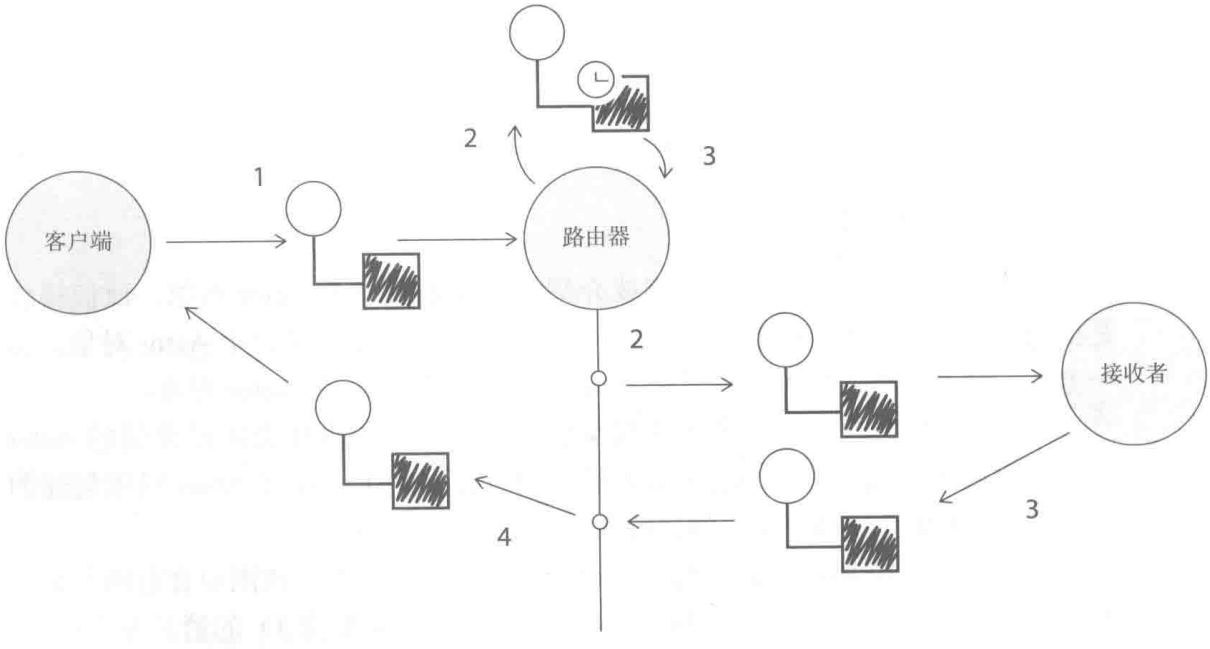

图 P.10　由顺序编号代表的消息发送次序和并发处理情况。

致谢

首先，我要感谢 Addison-Wesley 出版社选择并出版本书。我很荣幸能够再次和编辑 Chris Guzikowski 和 Chris Zahn 一起工作。我特别感谢 Chris Guzikowski，当本书不得不随 Akka 工具集的更改而大规模修改时他给予了极大的耐心。最后，我要说我们的努力是值得的。

如果没有 Carl Hewitt 博士的贡献，一本介绍 Actor 模型的书会变成什么样呢？Carl Hewitt 博士和他的同行们为整个世界贡献了一种独具匠心的计算模型，随着时间的推移该模型会越来越流行。

我还要感谢 Akka 开发团队为 Akka 工具集做出的出色贡献。Jonas Bonér 审阅了本书的多个章节，并作为 Akka 项目的创始人提出了他独特的观点。Akka 项目的技术主管 Roland Kuhn 也审阅了本书的部分章节，并为我提供了宝贵的反馈信息。我要感谢 Roland Kuhn 和 Jamie Allen，他们帮助我建立了本书介绍 Actor 模型的插图规范。此外，我还要感谢 Akka 团队的 Patrik Nordwall，他审阅了本书的前几章。

我特别感谢 Typesafe 公司的 Will Sargent 顾问，他为介绍 Akka 集群功能的章节做了许多贡献。尽管这部分内容很多，但 Will Sargent 从平凡之处找到了闪光点，并帮助我进一步提高它们的品质。

Thomas Lockney 和 Duncan DeVore 是本书两位前期的审稿人。Thomas Lockney 本身就是一位介绍 Akka 框架书籍的作者，Duncan DeVore 在帮助我审阅本书时，自己也正在撰写一本介绍 Akka 框架的书。我特别感谢 Thomas Lockney，当我在本书的前 3 章中进行一些早期尝试时，他给予我许多宽容。坦白地讲，Thomas Lockney 在反复审阅本书和对部分内容进行改进时体现的耐心，令我感到惊讶。

其他为提高本书品质做出贡献的审稿人包括：Idar Borlaug、Brian Dunlap、Tom Janssens、Dan Bergh Johnsson、Tobias Neef、Tom Stockton 和 Daniel Westheide。感谢大家为本书提供的宝贵反馈信息，这些信息使本书能够百尺竿头更进一步。我还特别感谢 Daniel Westheide，他就像一个人体的 Scala 编译器，从本书的示例代码中找出了许多难以发现的错误。