

# Netty

## 实战

# Netty

IN ACTION

[美] Norman Maurer  
Marvin Allen Wolfthal 著  
何品 译



中国工信出版集团



人民邮电出版社  
POSTS & TELECOM PRESS



# Netty 实战

Netty  
IN ACTION

〔美〕 Norman Maurer 著  
Marvin Allen Wolfthal  
何品 译

人 民 邮 电 出 版 社

## 图书在版编目（C I P）数据

Netty实战 / (美) 诺曼·毛瑞尔, (美) 马文·艾伦·沃尔夫泰尔著 ; 何品译. -- 北京 : 人民邮电出版社, 2017.6

书名原文: Netty in Action  
ISBN 978-7-115-45368-6

I. ①N… II. ①诺… ②马… ③何… III. ①JAVA语言—程序设计 IV. ①TP312. 8

中国版本图书馆CIP数据核字(2017)第075894号

---

◆! 著 [美] Norman Maurer Marvin Allen Wolfthal  
译 何 品  
责任编辑 杨海玲  
责任印制 焦志炜  
◆! 人民邮电出版社出版发行!!!! 北京市丰台区成寿寺路 11 号  
邮编 100164 电子邮件 315@ptpress.com.cn!  
网址 <http://www.ptpress.com.cn/>  
三河市海波印务有限公司印刷  
◆! 开本: 800×1000 1/16  
印张: 17.25  
字数: 362 千字 2017 年 6 月第 1 版  
印数: 1 - 3 000 册 2017 年 6 月河北第 1 次印刷  
著作权合同登记号 图字: 01-2015-8782 号!

---

定价: 69.00 元

读者服务热线: (010)81055410 印装质量热线: (010)81055316!  
广告经营许可证: 京东工商广字第 8052 号

# 内容提要

---

本书是为想要或者正在使用 Java 从事高性能网络编程的人而写的，循序渐进地介绍了 Netty 各个方面的内容。

本书共分为 4 个部分：第一部分详细地介绍 Netty 的相关概念以及核心组件，第二部分介绍自定义协议经常用到的编解码器，第三部分介绍 Netty 对于应用层高级协议的支持，会覆盖常见的协议及其在实践中的应用，第四部分是几个案例研究。此外，附录部分还会简单地介绍 Maven，以及如何通过使用 Maven 编译和运行本书中的示例。

阅读本书不需要读者精通 Java 网络和并发编程。如果想要更加深入地理解本书背后的理念以及 Netty 源码本身，可以系统地学习一下 Java 网络编程、NIO、并发和异步编程以及相关的设计模式。



# 中文版序

---

现代互联网架构，分布式系统是一个绕不开的话题。一款优秀的网络通信框架将在分布式系统的构建中起到举足轻重的作用。其中，特别出名的有 SUN 公司的 Grizzly 框架、JBoss 的 XIO、Apache 的 MINA 以及赫赫有名也是使用最广泛的 Netty 框架。

需要指出的是，网络通信框架的优秀不仅仅体现在性能和效率上，更重要的体现是，是否能够屏蔽底层复杂度，编程模型是否简单易懂，是否适用更多的应用场景，以及开发社区是否活跃。Netty 的成功正是很好地满足了上述的这几点。作为互联网从业人员，熟悉基于 Netty 网络编程乃至深入理解 Netty 的设计和实现，对于无论是自研系统，还是学习开源产品，都有很大的帮助。

网络上介绍、分析 Netty 的中文文章不少，其中能够做到成体系介绍，深入浅出，原理应用并重的寥寥。Manning 出版社的《Netty in Action》是一本出色的 Netty 教程。通过对这本书的学习，读者可以快速掌握基于 Netty 的编程，以及框架背后的设计哲学。可惜一直没有国内出版社引进出版中文版，像我这样的英文苦手，只能硬着头皮去啃英文版本，不仅学得慢，有些章节还不能很好地领会作者的意图。

很高兴地得知这本经典著作要在国内出版中文版，并且是由对 Netty 研究很深的工程师——何品——翻译的。我和何品打过几次交道，深入探讨过分布式架构以及网络通信框架方面的话题，受益良多。同时，也很惊讶于何品对技术的痴迷，以及他的技术深度和广度。诚挚地邀请他加入我们团队未果，甚为遗憾。十分期待这本书能很快出版发行，相信本书中文版的出版对投身互联网系统开发的工程师快速掌握 Netty 会有很大的帮助。

罗毅  
阿里巴巴中间件技术部高级技术专家



# 译者序

---

我对于 Netty 的接触始于 2012 年的工作，那时需要处理一些自定义协议相关的内容，对于技术的热情激发了我对于 Netty 源代码的学习，并促使我后续更加系统地学习了很多相关的知识。但是苦于缺乏相关中文资料以及系统性的指导，使得我在最终能够看懂 Netty 源代码并且为 Netty 项目做出贡献之前，花费了大量的时间，走了很多的弯路，这样的弯路自然也是充满苦楚和寂落的。

在后来又接触到了 Play 和 Akka，并且在得知了这些高性能网络编程和并发框架的底层正是基于 Netty 的时候，更是让我肯定了自己过去的投入，Netty FTW！那时，正值 Netty 4 重写，从源头改善了很多问题，提供了更好的并发模型，进一步降低了 GC 消耗。在跟进 Netty 4 的开发过程的同时，我也不断地丰富自己的知识和经验，并开启了我后续职业生涯的大门。再后来，当我得知 Norman 正在编写一本关于 Netty 的书的时候，非常激动，最终得以读到本书的 MEAP 版本，并能够有幸参与这本书的翻译工作。

这本书循序渐进、系统性地讲解了 Netty 的各个组件，以及其背后的设计哲学，并且对于想要深入理解 Netty 源代码的读者给出了相应的指导。难能可贵的是，这本书还附带了 5 个由行业一线公司撰写的 Netty 在实践中的案例研究，并贴心地准备了一个 Maven 相关的介绍。

本书的翻译经历了两个夏天和两个冬天（MEAP 版开始同步翻译）。为了能给大家呈现一个尽可能完善的中文版译本，我尽可能地使用了最新的原版书稿，并就书中的内容和原作者进行了积极的沟通。但是碍于个人水平有限，一些纰漏还请大家通过 <https://github.com/ReactivePlatform/netty-in-action-cn> 和我取得联系，也欢迎大家与我讨论书中代码清单相关的问题。

最后，我要感谢本书的编辑的耐心和悉心指导，感谢帮我牵线的 InfoQ 的臧秀涛，以及帮我审读了这本书的朋友们。当然，还要感谢我的家人，在他们的支持和理解下，这本书才得以完成，并呈现在大家的面前。

# 译者简介

---



**何品** 目前是淘宝的一名资深软件工程师，热爱网络、并发、异步相关的主题以及函数式编程，同时也是 Netty、Akka 等项目的贡献者，活跃于 Scala 社区，目前也在从事 GraphQL 相关的开发工作。

# 序

---

曾经人们认为 Web 应用服务器将会让我们忘记如何编写 HTTP 或者 RPC 服务器。不幸的是，这个白日梦并没有持续多久。我们正在处理的负载量以及功能变化的速度一直在不断地增加，超出了传统的三层体系结构的承受能力，我们正被迫将应用程序切分成很多块，并分发到更大的机器集群中。

运行一个如此庞大的分布式系统引发了两个有趣的问题：运行成本和延迟。如果我们将单个节点的性能提高 30%，或者甚至超过 100%，那么我们可以节省多少台机器呢？当一个来自 Web 浏览器的查询触发了几十个跨越了很多不同机器的内部远程过程调用时，我们如何能达到最低的延迟呢？

在本书（第一本关于 Netty 项目的书）中，Norman Maurer（Netty 的核心贡献者之一）通过展示如何使用 Netty 构建高性能、低延迟的网络应用程序，给出了这些问题的最终答案。读完这本书，你就能够构建所有可能的网络应用程序了，从轻量级的 HTTP 服务器到高度定制化的 RPC 服务器。

本书之所以能令人印象深刻，一方面是因为它是由知晓 Netty 每个细节的核心贡献者编写的，另一方面是因为它包含了几家在其生产系统中使用了 Netty 的公司（Twitter、Facebook 和 Firebase 等）的案例研究。我相信，通过展示这些使用它们的公司是如何能够释放他们基于 Netty 的应用程序的能力的，这些案例研究将会启迪你。

你可能会惊奇地发现，早在 2001 年，Netty 只是我的个人项目，当时我是一名本科生 (<http://t.motd.kr/ko/archives/1930>)，而今天这个项目仍然还在并且还充满了活力，感谢像 Norman 这样的热心的贡献者们，他们花了许多个不眠之夜来致力于该项目 (<http://netty.io/community.html>)。我希望通过鼓励本书的读者来贡献项目，开启该项目的另一个篇章，继续“开启网络编程的未来”。

Trustin Lee  
Netty 项目创始人



# 前言

---

回首过去，我仍然不敢相信我做到了。

当我从 2011 年年末开始为 Netty 做贡献时，我怎么也想不到我会写一本关于 Netty 的书，并且成为该框架本身的核心开发者之一。

这一切都始于我在 2009 年参与的 Apache James 项目，一个在 Apache 软件基金会下开发的基于 Java 的邮件服务器。

像许多应用程序一样，Apache James 需要构建在一个坚实的网络抽象之上。在考察提供网络抽象的项目领域时，我偶然地发现了 Netty，并且立即就爱上了它。在我从用户的角度更加地熟悉了 Netty 之后，我便开始转向改进它和回馈社区。

尽管我第一次贡献的范围有限，但是很快变得明显的是，进行贡献以及和社区进行相关问题的讨论，尤其是和项目的创始人 Trustin Lee，对于我的个人成长非常有益。这样的经验牢牢地吸引了我，我喜欢将我的空闲时间更多地投入到社区中。我在邮件列表上提供帮助，并且加入了 IRC 频道的讨论。致力于 Netty 开始是一种爱好，但很快就演变成了一种激情。

我对 Netty 的激情最终导致我在 Red Hat 就业。这简直是美梦成真，因为 Red Hat 雇佣我来致力于我所热爱的项目。我最终知道了 Claus Ibsen 在那时正（现在仍然）致力于 Apache Camel。Claus 和我认识到，虽然 Netty 拥有坚实的用户基础以及良好的 JavaDoc，但是它缺乏一个更加高级别的文档。Claus 是《Camel in Action》( Manning, 2010 ) 的作者，他给了我为 Netty 写一本类似的想法。关于这个想法，我考虑了几个星期，最终接受了。这也就有了本书。

在编写本书的过程中，我也越来越多地参与到了社区中。伴随着超过 1000 次的提交<sup>①</sup>，我最终成为了仅次于 Trustin Lee 的最活跃的贡献者。我经常在世界各地的各种会议以及技术聚会上演讲 Netty。最终 Netty 开启了另一个在苹果公司的就业机会，我目前在云基础设施工程团队

---

<sup>①</sup> 截至中文版出版前，已经超过 2000 次提交了。——译者注

(Cloud Infrastructure Engineering Team) 担任资深软件工程师。我继续致力于Netty，并且经常贡献回馈社区，同时也帮助推动该项目。

Norman Maurer  
苹果公司，云基础设施工程

我在马萨诸塞州韦斯顿的 Harvard Pilgrim Health Care 担任 Dell Services 的顾问时，就主要侧重于构建可复用的基础设施组件。我们的目标是找到这样一种扩展通用代码库的方式：它不仅对通常的软件过程有利，而且还能将应用程序开发者从编写既麻烦又平凡的管道代码（plumbing code）的责任中解脱出来。

我一度发现，有两个相关的项目都在使用一个第三方的理赔处理系统，该系统只支持直接的 TCP/IP 通信。其中一个项目需要使用 Java 重新实现一个文档不太详细的构建在供应商的专有的基于分隔的格式上的遗留 COBAL 模块。这个模块最终被另一个项目取代了，那个项目将使用较新的基于 XML 的接口来连接到该相同理赔系统上。（但是使用的仍然是裸套接字，而不是 SOAP！）

在我看来，这是一个理想的开发一个通用 API 的机会，而且也充满了乐趣。我知道将会有严格的吞吐量和可靠性要求，并且设计也仍然在不断地演进。显然，为了支持快速的迭代周期，底层的网络代码必须完全和业务逻辑解耦。

我对于 Java 的高性能网络编程框架的调研把我直接带到了 Netty 面前。（在第 1 章开头读者会读到一个假设的项目，它其实基本上取材自现实生活。）我很快就确信了 Netty 的方式，使用可动态配置的编码器和解码器，能够完美地满足我们的需求：两个项目将可以使用相同的 API，并部署所使用的特定数据格式所需的处理器。在我发现该供应商的产品也是基于 Netty 的之后，我变得更加坚信了！

就在那时，我得知有一本我一直都在期待的叫《Netty 实战》的书正在编写中。我读了早期的书稿，并带着一些问题和建议很快和 Norman 取得了联系。在我们多次的交流过程中，我们常常会谈到要记住最终用户的视角，而且因为我当时正在参与一个实实在在的 Netty 项目，所以我很高兴地担当了这个（合著者/最终用户）角色。

我希望，通过这种方式，我们能够成功地满足开发者们的需求。如果您有任何关于我们如何能够使得本书变得更加有用的建议，请在 <https://forums.manning.com/forums/netty-in-action> 联系我们。

Marvin Allen Wolfthal  
Dell Services

# 致谢

---

Manning 团队使得编写本书的过程很快乐，而且他们从未曾在这份快乐比所预期的要长一些时有过任何抱怨。从 Mike Stephens（他使得这一切成为了可能）和 Jeff Bleiel（从他那里我们学到了一些关于协作的新知识）到 Jennifer Stout、Andy Carroll 和 Elizabeth Martin（她展示出的冷静和耐心令我们望尘莫及），他们所有人都具备非常的专业水平和素质，激励着作者们也尽善尽美。

感谢那些帮助校审本书的人们，不管是通过阅读那些早期版本的章节并在 Author Online 论坛张贴勘误的，还是通过在本书编写的各个阶段校审书稿的。你们是本书的一部分，应该感到自豪。没有你们，这本书将不可能会像现在这样。特别地感谢下面的这些审阅者们：Achim Friedland、Arash Bizhan Zadeh、Bruno Georges、Christian Bach、Daniel Beck、Declan Cox、Edward Ribeiro、Erik Onnen、Francis Marchi、Gregor Zurowski、Jian Jin、Jürgen Hoffmann、Maksym Prokhorenko、Nicola Grigoletti、Renato Felix 和 Yestin Johnson。同样也感谢我们优秀的技术校对：David Dossot 和 Neil Rutherford。

我们非常感激并且由衷地感谢 Bruno de Carvalho、Sara Robinson、Greg Soltis、Erik Onnen、Andrew Cox 以及 Jeff Smick，他们贡献了第 14 章和第 15 章的案例研究。

最后但并非最不重要，感谢所有支持 Netty 以及开源项目的人们，没有你们和社区，就不可能有这个项目。通过社区，我们得以结识新朋友、在世界各地的会议上讨论，并且同时获得了专业和个人方面的成长。

## Norman Maurer

我想要感谢我的前同事也是朋友 Jürgen Hoffmann（也叫 Buddy）。Jürgen 帮助我找到了我进入开源世界的道路，并且向我展示了当你拥有足够的勇气参与时，你将能构建出多么酷的东西。如果没有他，我可能永远也不会接触到编程，也因此不会发现我真正的专业激情所在。

另外，要非常地感谢我的朋友 Trustin Lee——Netty 的创始人，最初是他帮助并且鼓励了我为 Netty 项目做贡献，还为我们的书作序。我很荣幸能够认识你并能够和你成为朋友！我相信通过继续一起工作，Netty 将继续令人惊叹并长久存在！

我还想感谢我的合著者 Marvin Wolfthal。尽管 Marvin 在该项目的后期才加入，但他帮助我极大地提高了整体的结构和内容。没有他，这本书不可能有现在的样子。这让我想到了 Manning 团队本身，他们总是能够给予帮助和正确的指引，使得编写一本书的想法成为现实。

感谢我的父母 Peter 和 Christa 一直以来支持我以及我的想法。

最要感谢的是我的妻子 Jasmina 以及我的孩子们 Mia Soleil 和 Ben，感谢他们在我编写这本书的过程中所给予的支持。没有你们就不可能有这本书。

## Marvin Wolfthal

首先，我要感谢我的合著者 Norman Maurer，感谢他出色的工作以及他的友善。虽然我加入该项目的时间比较晚，但他依然让我感觉好像是从第一天开始就成为了它的一部分似的。

对于我过去和现在在 Dell Services 以及 Harvard Pilgrim Health Care 的同事们，我衷心地感谢他们的帮助及鼓励。他们创造了不可多得的环境，在那里，不仅可以表达新的想法，而且还能将其筑成现实。致 Deborah Norton、Larry Rapisarda、Dave Querusio、Vijay Bhatt、Craig Bogovich 以及 Sharath Krishna，特别要感谢的是他们的支持，以及更难得的是他们的信任——我相信没有多少软件开发者能够被给予我在过去 4 年里所享受到的创造性的机会，包括将 Netty 引入到我们的工具集中。

但最重要的是，感谢我心爱的妻子 Katherine，她让我永远不会忘记那些真正重要的东西。

# 关于本书

---

Netty 是一款用于快速开发高性能的网络应用程序的 Java 框架。它封装了网络编程的复杂性，使网络编程和 Web 技术的最新进展能够被比以往更广泛的开发人员接触到。

Netty 不只是一个接口和类的集合；它还定义了一种架构模型以及一套丰富的设计模式。但是直到现在，依然缺乏一个全面的、系统性的用户指南，已经成为入门 Netty 的一个障碍，这种情况也是本书旨在改变的。除了解释该框架的组件以及 API 的详细信息之外，本书还会展示 Netty 如何能够帮助开发人员编写更高效的、可复用的、可维护的代码。

## 谁应该阅读本书

本书假定读者熟悉中等级别的 Java 主题，如泛型和多线程处理。不要求有高级网络编程的经验，但是熟悉基本的 Java 网络编程 API 将大有裨益。

Netty 使用 Apache Maven 作为它的构建管理工具。如果读者还未使用过 Maven，那么附录将会为读者提供运行本书示例代码所需要的信息。读者也可以复用这些示例的 Maven 配置，作为自己的基于 Netty 的项目的起点。

## 导读

本书共分 4 个部分，且有一个附录。

### 第一部分：Netty 的概念及体系结构

第一部分是对框架的详细介绍，涵盖了它的设计、组件以及编程接口。

第 1 章首先简要概述了阻塞和非阻塞的网络 API，以及它们对应的 JDK 接口。我们引入 Netty 作为构建高度可伸缩的、异步的、事件驱动的网络编程应用的工具包。我们将首先看一下该框架的基础构件块：Channel、回调、Future、事件及 ChannelHandler。

第 2 章解释了如何配置读者的系统以构建并运行本书中的示例代码。我们将用一个简单的应

用程序来测试它，这是一个回送从连接的客户端接收到的消息的服务器应用程序。我们还介绍了引导 (Bootstrap) ——在运行时组装和配置一个应用程序的所有组件的过程。

第 3 章首先讨论了 Netty 的技术以及体系结构方面的内容。介绍了该框架的核心组件：Channel、EventLoop、ChannelHandler 以及 ChannelPipeline。这一章的最后解释了引导服务器和客户端之间的差异。

第 4 章讨论了网络传输，并且对比了通过 JDK API 和 Netty 使用阻塞和非阻塞传输的用法。我们研究了 Netty 的传输 API 的底层接口的层级关系以及它们所支持的传输类型。

第 5 章专门介绍了该框架的数据处理 API——ByteBuf，Netty 的字节容器。我们描述了它相对于 JDK 的 ByteBuffer 的优势，以及如何分配和访问由 ByteBuf 所使用的内存。我们展示了如何通过引用计数来管理内存资源。

第 6 章重点介绍了核心组件 ChannelHandler 和 ChannelPipeline，它们负责调度应用程序的处理逻辑，并驱动数据和事件经过网络层。其他的主题包括在实现高级用例时 ChannelHandlerContext 的角色，以及在多个 ChannelPipeline 之间共享 ChannelHandler 的缘由。这一章的最后说明了如何处理由入站事件和出站事件所触发的异常。

第 7 章提供了关于线程模型的一般概述，并详细地介绍了 Netty 的线程模型。我们研究了 interface EventLoop，它是 Netty 的并发 API 的主要部分，并解释了它和线程以及 Channel 的关系。这个信息对于理解 Netty 是如何实现异步的、事件驱动的网络编程模型来说至关重要。我们展示了如何通过 EventLoop 进行任务调度。

第 8 章以介绍 Bootstrap 类的层级结构作为引子，深入地讲解了引导。我们重新审视了一些基本用例以及一些特殊用例，例如，在一个服务器应用程序中引导一个客户端连接、引导数据报 Channel，以及在引导的过程中添加多个 ChannelHandler。这一章最后讨论了如何优雅地关闭应用程序并有序地释放所有的资源。

第 9 章是关于对 ChannelHandler 进行单元测试的讨论，对此 Netty 提供了一个特殊的 Channel 实现——EmbeddedChannel。本章的示例展示了如何使用这个类和 JUnit 一起来测试入站和出站 ChannelHandler 实现。

## 第二部分：编解码器

数据转换是网络编程中最常见的操作之一。第二部分介绍了 Netty 提供的用于简化这一任务的丰富的工具集。

第 10 章首先解释了解码器和编码器，它们将字节序列从一种格式转换为另外一种格式。一个无处不在的例子便是将一个非结构化的字节流转换为一个特定于协议的布局结构，或者相反的。编解码器则是一个结合了编码器以及解码器以处理双向转换的组件。我们提供了几个例子，展示了通过 Netty 的编解码器框架类创建自定义的解码器以及编码器是多么地容易。

第 11 章研究了 Netty 提供的用于各种用例的编解码器以及 ChannelHandler。这些类包括用于协议的（如 SSL/TLS、HTTP/HTTPS、WebSocket 以及 SPDY）即用型的编解码器，以及能

够通过扩展来处理几乎任意的基于分隔符的协议、变长协议或者定长协议的解码器。这一章的最后介绍了用于写入大型数据的和用于序列化的框架组件。

## 第三部分：网络协议

第三部分详细阐述了几种本书前面简要介绍过的网络协议。我们将会再次看到 Netty 是如何使你能在自己的应用程序中轻松采用复杂的 API，而又不必关心其内部复杂性的。

第 12 章展示了如何使用 WebSocket 协议来实现 Web 服务器和客户端之间的双向通信。示例程序是一个聊天室服务器，其允许所有已连接的用户与其他已连接的用户进行实时通信。

第 13 章通过利用了用户数据报协议（UDP）的广播能力的服务器和客户端应用程序，说明了 Netty 对于无连接协议的支持。如同前面的那些示例一样，我们使用了一组特定于协议的支持类：DatagramPacket 和 NioDatagramChannel。

## 第四部分：案例研究

第四部分介绍了由使用 Netty 实现了任务关键型系统的知名公司提交的 5 份案例研究。这些案例不仅说明了我们在整本书中所讨论过的框架各个组件在现实世界中的应用，而且还演示了 Netty 的设计以及架构原则，在构建高度可伸缩和可扩展的应用程序方面的应用。

第 14 章有 Droplr、Firebase 以及 Urban Airship 提交的案例研究。

第 15 章有 Facebook 和 Twitter 提交的案例研究。

## 附录：Maven 介绍

该附录的主要目的是提供一个对于 Apache Maven 的基本介绍，以便读者可以编译和运行本书的示例代码清单，并在开始使用 Netty 时扩展它们来创建自己的项目。

介绍了以下主题：

- Maven 的主要目标和用途；
- 安装以及配置 Maven；
- Maven 的基本概念——POM 文件、构件、坐标、依赖、插件及存储库；
- Maven 配置的示例，POM 的继承以及聚合；
- Maven 的命令行语法。

## 代码约定和下载

这本书提供了丰富的示例，说明了如何利用每个涵盖的主题。为了将代码和普通文本区分开，代码清单或者正文中的代码都是以等宽字体（如 fixed-width font like this）显示的。此外，正文中的类和方法名、对象属性以及其他代码相关的术语和内容也都以等宽字体呈现。

偶尔，代码是斜体的，如 `reference.dump()`。在这种情况下，不要逐字输入 `reference`,

要把它替换为所需的内容。

本书的源代码可以从出版商的网站[www.manning.com/books/netty-in-action](http://www.manning.com/books/netty-in-action)以及GitHub的项目地址<https://github.com/normanmaurer/netty-in-action>获取<sup>①</sup>。我们将源代码构造成一个多模块的Maven项目，其中包含一个顶级POM和多个对应于本书各章的模块。

## 关于作者

**Norman Maurer**<sup>②</sup>是Netty的核心开发人员之一，Apache软件基金会的一员。在过去的几年，他还是很多开源项目的贡献者。他是Apple公司的一名资深软件工程师，Netty和其他网络相关的项目是他在iCloud团队的工作内容。

**Marvin Wolfthal**<sup>③</sup>作为开发者、架构师、讲师和作者一直活跃在多个软件开发领域。他很早就开始使用Java，并且协助Sun开发了它第一批致力于促进分布式对象技术的程序。作为这些努力的一部分，他使用C++、Java和CORBA为Sun Education编写了第一套跨语言的编程课程。从那时起，他的主要关注点就一直是中间件的设计和开发，主要针对金融行业。他目前是Dell Services的一名顾问，致力于将Java世界中产生的方法论拓展到其他的企业计算领域中，例如，将持续集成的实践应用到数据库的开发中。Marvin还是钢琴家和作曲家，他的作品已由维也纳的Universal Edition公司发行<sup>④</sup>。他和他的妻子凯瑟琳以及他们的3只猫伙伴Fritz、Willy和Robbie住在马萨诸塞州的韦斯顿。

## 作者在线

购买本书的读者可以免费访问Manning出版社运营的一个私有Web论坛<sup>⑤</sup>，在那里，可以评论本书、提技术问题，还可以获得作者和其他用户的帮助。如果要访问或者订阅该论坛，可以用Web浏览器访问[www.manning.com/books/netty-in-action](http://www.manning.com/books/netty-in-action)。这个页面提供了以下信息：注册之后如何访问论坛；可以获得什么样的帮助；该论坛的一些行为准则；本书示例的源代码的链接、勘误表以及其他下载资源。

Manning 承诺为我们的读者提供一个交流场所，在那里读者之间以及读者和作者之间可以进行有意义的对话。但是对于作者方面的参与并没有做任何数量上的承诺，作者对于作者在线（AO）的贡献仍然是自愿的（和无偿的）。我们建议你向作者提一些富有挑战性的问题，以免他们没兴趣回答！

只要这本书尚未绝版，就可以从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。

---

① 本书中文版的源代码可以从GitHub的项目地址<https://github.com/ReactivePlatform/netty-in-action-cn>获取，也可以在异步社区（[www.epubit.com.cn](http://www.epubit.com.cn)）本书页面下载。——译者注

② Norman Maurer 的个人网站是<http://normanmaurer.me/>，在这里可以找到更多关于 Netty 的讨论。——译者注

③ Marvin Wolfthal 的个人网站是<http://www.weichi.com/maw/>。——译者注

④ 唱片的在线试听地址是<http://www.universaledition.com/composers-and-works/Marvin-Wolfthal/composer/4038>。——译者注

⑤ 本书中文版的读者也可以访问本书在异步社区的相应页面。——译者注

# 关于封面插图

---

本书封面上的插画名为“卢森堡地区的居民”(A Resident of the Luxembourg Quarter)。该插画选自多位艺术家的19世纪作品集，由Louis Curmer编辑，并于1841年在巴黎出版。该作品集的标题是《Les Français peints par eux-mêmes》，翻译过来是“法国人民的自画像”。每幅插画都是手工精细绘制和着色的，作品集中丰富多样的作品向我们生动地展现了200年前世界上各个区域、城镇、村庄以及居民区的文化是多么迥异。人们彼此分开，讲不同的方言和语言。仅仅通过他们的服饰就能够很容易地辨别出他们在哪儿生活，住在城镇里还是住在乡下、干什么工作或者有什么样的生活地位。

自那以后，服饰的风格已然发生了变化，当时各地如此丰富多样的风格已经逐渐消失。现在已经很难分辨不同大洲的居民，更别说区分不同城镇或者地区的居民了。也许我们使用的多样性换取了更加多样化的个人生活——当然也是更加多样化和快节奏的科技生活。

在很难将一本计算机图书与另一本区分开的时代，Manning通过使用基于两个世纪以前的多样化的区域生活的图书封面，让作品集中的插画重现于世，比如这一幅，借以来赞美计算机行业的创造力和进取精神。



# 目录

---

## 第一部分 Netty 的概念及体系结构

1 第1章 Netty——异步 和事件驱动	3
1.1 Java 网络编程	4
1.1.1 Java NIO	5
1.1.2 选择器	6
1.2 Netty 简介	6
1.2.1 谁在使用 Netty	7
1.2.2 异步和事件驱动	8
1.3 Netty 的核心组件	9
1.3.1 Channel	9
1.3.2 回调	9
1.3.3 Future	10
1.3.4 事件和 Channel Handler	11
1.3.5 把它们放在一起	12
1.4 小结	13

2 第2章 你的第一款 Netty 应用程序	14
2.1 设置开发环境	14
2.1.1 获取并安装 Java 开发 工具包	14
2.1.2 下载并安装 IDE	15
2.1.3 下载和安装 Apache Maven	15
2.1.4 配置工具集	16

2.2 Netty 客户端/服务器 概览	16
2.3 编写 Echo 服务器	17
2.3.1 ChannelHandler 和 业务逻辑	17
2.3.2 引导服务器	18
2.4 编写 Echo 客户端	21
2.4.1 通过 ChannelHandler 实现 客户端逻辑	21
2.4.2 引导客户端	22
2.5 构建和运行 Echo 服务器 和客户端	24
2.5.1 运行构建	24
2.5.2 运行 Echo 服务器和 客户端	27
2.6 小结	29
3 第3章 Netty 的组件和设计	30
3.1 Channel、EventLoop 和 ChannelFuture	30
3.1.1 Channel 接口	31
3.1.2 EventLoop 接口	31
3.1.3 ChannelFuture 接口	32
3.2 ChannelHandler 和 ChannelPipeline	32
3.2.1 ChannelHandler 接口	32
3.2.2 ChannelPipeline 接口	33
3.2.3 更加深入地了解 ChannelHandler	34
3.2.4 编码器和解码器	35
3.2.5 抽象类 SimpleChannel- InboundHandler	35

**4****第4章 传输 38**

- 4.1 案例研究：传输迁移 38
  - 4.1.1 不通过 Netty 使用 OIO 和 NIO 39
  - 4.1.2 通过 Netty 使用 OIO 和 NIO 41
  - 4.1.3 非阻塞的 Netty 版本 42
- 4.2 传输 API 43
- 4.3 内置的传输 45
  - 4.3.1 NIO——非阻塞 I/O 46
  - 4.3.2 Epoll——用于 Linux 的本地非阻塞传输 47
  - 4.3.3 OIO——旧的阻塞 I/O 48
  - 4.3.4 用于 JVM 内部通信的 Local 传输 48
  - 4.3.5 Embedded 传输 49
- 4.4 传输的用例 49
- 4.5 小结 51

**5****第5章 ByteBuf 52**

- 5.1 ByteBuf 的 API 52
- 5.2 ByteBuf 类—Netty 的数据容器 53
  - 5.2.1 它是如何工作的 53
  - 5.2.2 ByteBuf 的使用模式 53
- 5.3 字节级操作 57
  - 5.3.1 随机访问索引 57
  - 5.3.2 顺序访问索引 57
  - 5.3.3 可丢弃字节 58
  - 5.3.4 可读字节 58
  - 5.3.5 可写字节 59
  - 5.3.6 索引管理 59
  - 5.3.7 查找操作 60
  - 5.3.8 派生缓冲区 60
  - 5.3.9 读/写操作 62
  - 5.3.10 更多的操作 64
- 5.4 ByteBufHolder 接口 65
- 5.5 ByteBuf 分配 65
  - 5.5.1 按需分配：ByteBufAllocator 接口 65
  - 5.5.2 Unpooled 缓冲区 67
  - 5.5.3 ByteBufUtil 类 67

**6****第6章 ChannelHandler 和 ChannelPipeline 70**

- 6.1 ChannelHandler 家族 70
  - 6.1.1 Channel 的生命周期 70
  - 6.1.2 ChannelHandler 的生命周期 71
  - 6.1.3 ChannelInboundHandler 接口 71
  - 6.1.4 ChannelOutboundHandler 接口 73
  - 6.1.5 ChannelHandler 适配器 74
  - 6.1.6 资源管理 74
- 6.2 ChannelPipeline 接口 76
  - 6.2.1 修改 ChannelPipeline 78
  - 6.2.2 触发事件 79
- 6.3 ChannelHandlerContext 接口 80
  - 6.3.1 使用 ChannelHandlerContext 82
  - 6.3.2 ChannelHandler 和 Context 的高级用法 84
- 6.4 异常处理 86
  - 6.4.1 处理入站异常 86
  - 6.4.2 处理出站异常 87
- 6.5 小结 88

**7****第7章 EventLoop 和线程模型 89**

- 7.1 线程模型概述 89
- 7.2 EventLoop 接口 90
  - 7.2.1 Netty 4 中的 I/O 和事件处理 92
  - 7.2.2 Netty 3 中的 I/O 操作 92
- 7.3 任务调度 93
  - 7.3.1 JDK 的任务调度 API 93
  - 7.3.2 使用 EventLoop 调度任务 94
- 7.4 实现细节 95
  - 7.4.1 线程管理 95
  - 7.4.2 EventLoop/线程的分配 96
- 7.5 小结 98

**8****第 8 章 引导** 99

- 8.1 Bootstrap 类 99
- 8.2 引导客户端和无连接协议 101
  - 8.2.1 引导客户端 102
  - 8.2.2 Channel 和 EventLoop-Group 的兼容性 103
- 8.3 引导服务器 104
  - 8.3.1 ServerBootstrap 类 104
  - 8.3.2 引导服务器 105
- 8.4 从 Channel 引导客户端 107
- 8.5 在引导过程中添加多个 ChannelHandler 108
- 8.6 使用 Netty 的 Channel-Option 和属性 110
- 8.7 引导 DatagramChannel 111
- 8.8 关闭 112
- 8.9 小结 112

**9****第 9 章 单元测试** 113

- 9.1 EmbeddedChannel 概述 113
- 9.2 使用 EmbeddedChannel 测试 ChannelHandler 115
  - 9.2.1 测试入站消息 115
  - 9.2.2 测试出站消息 118
- 9.3 测试异常处理 119
- 9.4 小结 121

**第二部分 编解码器****10****第 10 章 编解码器框架** 125

- 10.1 什么是编解码器 125
- 10.2 解码器 125
  - 10.2.1 抽象类 ByteTo-MessageDecoder 126
  - 10.2.2 抽象类 ReplayingDecoder 127
  - 10.2.3 抽象类 Message-ToMessageDecoder 128

- 10.2.4 TooLong-FrameException 类 130
- 10.3 编码器 131
  - 10.3.1 抽象类 Message-ToByteEncoder 131
  - 10.3.2 抽象类 Message-ToMessageEncoder 132
- 10.4 抽象的编解码器类 133
  - 10.4.1 抽象类 Byte-ToMessageCodec 133
  - 10.4.2 抽象类 Message-ToMessageCodec 134
  - 10.4.3 CombinedChannel-DuplexHandler 类 137
- 10.5 小结 138

**11****第 11 章 预置的 ChannelHandler 和编解码器** 139

- 11.1 通过 SSL/TLS 保护 Netty 应用程序 139
- 11.2 构建基于 Netty 的 HTTP/HTTPS 应用程序 141
  - 11.2.1 HTTP 解码器、编码器和编解码器 141
  - 11.2.2 聚合 HTTP 消息 143
  - 11.2.3 HTTP 压缩 144
  - 11.2.4 使用 HTTPS 145
  - 11.2.5 WebSocket 146
- 11.3 空闲的连接和超时 148
- 11.4 解码基于分隔符的协议 和基于长度的协议 150
  - 11.4.1 基于分隔符的协议 150
  - 11.4.2 基于长度的协议 153
- 11.5 写大型数据 155
- 11.6 序列化数据 157
  - 11.6.1 JDK 序列化 157
  - 11.6.2 使用 JBoss Marshalling 进行序列化 157
  - 11.6.3 通过 Protocol Buffers 序列化 159
- 11.7 小结 160

**第三部分 网络协议****12****第 12 章 WebSocket** 163

- 12.1 WebSocket 简介 163

12.2 我们的 WebSocket 示例 应用程序 164	14.2.2 长轮询 201 14.2.3 HTTP 1.1 keep-alive 和流水线化 204 14.2.4 控制 SslHandler 205 14.2.5 Firebase 小结 207
12.3 添加 WebSocket 支持 165 12.3.1 处理 HTTP 请求 165 12.3.2 处理 WebSocket 帧 168 12.3.3 初始化 Channel- Pipeline 169 12.3.4 引导 171	14.3 Urban Airship——构建 移动服务 207 14.3.1 移动消息的 基础知识 207 14.3.2 第三方递交 208 14.3.3 使用二进制 协议的例子 209 14.3.4 直接面向设备 的递交 211 14.3.5 Netty 擅长管理 大量的并发连接 212 14.3.6 Urban Airship 小结——跨 越防火墙边界 213
12.4 测试该应用程序 173	14.4 小结 214
12.5 小结 176	
<b>第 13 章 使用 UDP 广播事件 177</b>	<b>第 15 章 案例研究, 第二部分 215</b>
13.1 UDP 的基础知识 177	15.1 Netty 在 Facebook 的 使用: Nifty 和 Swift 215 15.1.1 什么是 Thrift 215 15.1.2 使用 Netty 改善 Java Thrift 的现状 216 15.1.3 Nifty 服务器的设计 217 15.1.4 Nifty 异步客户端的 设计 220 15.1.5 Swift: 一种更快的构建 Java Thrift 服务的方式 221 15.1.6 结果 221 15.1.7 Facebook 小结 224
13.2 UDP 广播 178	
13.3 UDP 示例应用程序 178	
13.4 消息 POJO: LogEvent 179	
13.5 编写广播者 180	
13.6 编写监视器 185	
13.7 运行 LogEvent- Broadcaster 和 LogEventMonitor 187	
13.8 小结 189	
<b>第四部分 案例研究</b>	
<b>第 14 章 案例研究, 第一部分 193</b>	
14.1 Droplr——构建 移动服务 193 14.1.1 这一切的起因 193 14.1.2 Droplr 是怎样工作的 194 14.1.3 创造一个更加快速 的上传体验 194 14.1.4 技术栈 196 14.1.5 性能 199 14.1.6 小结——站在 巨人的肩膀上 200	15.2 Netty 在 Twitter 的使用: Finagle 224 15.2.1 Twitter 成长的烦恼 224 15.2.2 Finagle 的诞生 224 15.2.3 Finagle 是如何 工作的 225 15.2.4 Finagle 的抽象 230 15.2.5 故障管理 231 15.2.6 组合服务 232 15.2.7 未来: Netty 232 15.2.8 Twitter 小结 233
14.2 Firebase——实时的 数据同步服务 200 14.2.1 Firebase 的架构 201	15.3 小结 233
	<b>附录 Maven 介绍 234</b>

# 第一部分

## Netty 的概念及体系结构

Netty 是一款用于创建高性能网络应用程序的高级框架。在第一部分，我们将深入地探究它的能力，并且在 3 个主要的方面进行示例：

- 使用 Netty 构建应用程序，你不必是一名网络编程专家；
- 使用 Netty 比直接使用底层的 Java API 容易得多；
- Netty 推崇良好的设计实践，例如，将你的应用程序逻辑和网络层解耦。

在第 1 章中，我们将首先小结 Java 网络编程的演化过程。在我们回顾了异步通信和事件驱动的处理的基本概念之后，我们将首先看一看 Netty 的核心组件。在第 2 章中，你将能够构建自己的第一款基于 Netty 的应用程序！在第 3 章中，你将开启对于 Netty 的细致探究之旅，从它的核心网络协议（第 4 章）以及数据处理层（第 5 章和第 6 章）到它的并发模型（第 7 章）。

我们将把所有的这些细节组合在一起，对第一部分进行总结。你将看到：如何在运行时配置基于 Netty 的应用程序的各个组件，以使它们协同工作（第 8 章），Netty 是如何帮助你测试你的应用程序的（第 9 章）。

# 第 1 章 Netty——异步和事件驱动



## 本章主要内容

- Java 网络编程
- Netty 简介
- Netty 的核心组件

假设你正在为一个重要的大型公司开发一款全新的任务关键型的应用程序。在第一次会议上，你得知该系统必须要能够扩展到支撑 150 000 名并发用户，并且不能有任何的性能损失，这时所有的目光都投向了你。你会怎么说呢？

如果你可以自信地说：“当然，没问题。”那么大家都会向你脱帽致敬。但是，我们大多数人可能会采取一个更加谨慎的立场，例如：“听上去是可行的。”然后，一回到计算机旁，我们便开始搜索“high performance Java networking”（高性能 Java 网络编程）。

如果你现在搜索它，在第一页结果中，你将会看到下面的内容：

### Netty: Home

[netty.io/](http://netty.io/)

Netty 是一款异步的事件驱动的网络应用程序框架，支持快速地开发可维护的高性能的面向协议的服务器和客户端。

如果你和大多数人一样，通过这样的方式发现了 Netty，那么你的下一步多半是：浏览该网站，下载源代码，仔细阅读 Javadoc 和一些相关的博客，然后写点儿代码试试。如果你已经有了扎实的网络编程经验，那么可能进展还不错，不然则可能是一头雾水。

这是为什么呢？因为像我们例子中那样的高性能系统不仅要求超一流的编程技巧，还需要几个复杂领域（网络编程、多线程处理和并发）的专业知识。Netty 优雅地处理了这些领域的知识，使得即使是网络编程新手也能使用。但到目前为止，由于还缺乏一本全面的指南，使得对它的学习过程比实际需要的艰涩得多——因此便有了这本书。

我们编写这本书的主要目的是：使得 Netty 能够尽可能多地被更加广泛的开发者采用。这也包

括那些拥有创新的内容或者服务，却没有时间或者兴趣成为网络编程专家的人。如果这适用于你，我们相信你将会非常惊讶自己这么快便可以开始创建你的第一款基于 Netty 的应用程序了。当然在另一个层面上讲，我们也需要支持那些正在寻找工具来创建他们自己的网络协议的高级从业人员。

Netty 确实提供了极为丰富的网络编程工具集，我们将花大部分的时间来探究它的能力。但是，Netty 终究是一个框架，它的架构方法和设计原则是：每个小点都和它的技术性内容一样重要，穷其精妙。因此，我们也将探讨很多其他方面的内容，例如：

- 关注点分离——业务和网络逻辑解耦；
- 模块化和可复用性；
- 可测试性作为首要的要求。

在这第1章中，我们将从一些与高性能网络编程相关的背景知识开始铺陈，特别是它在 Java 开发工具包（JDK）中的实现。有了这些背景知识后，我们将介绍 Netty，它的核心概念以及构建块。在本章结束之后，你就能够编写你的第一款基于 Netty 的客户端和服务器应用程序了。

## 1.1 Java 网络编程

早期的网络编程开发人员，需要花费大量的时间去学习复杂的 C 语言套接字库，去处理它们在不同的操作系统上出现的古怪问题。虽然最早的 Java (1995—2002) 引入了足够多的面向对象 façade (门面) 来隐藏一些棘手的细节问题，但是创建一个复杂的客户端/服务器协议仍然需要大量的样板代码 (以及相当多的底层研究才能使它整个流畅地运行起来)。

那些最早期的 Java API (`java.net`) 只支持由本地系统套接字库提供的所谓的阻塞函数。代码清单 1-1 展示了一个使用了这些函数调用的服务器代码的普通示例。

代码清单 1-1 阻塞 I/O 示例

```

    创建一个新的 ServerSocket, 用以
    监听指定端口上的连接请求
1 对 accept()方法的调
用将被阻塞, 直到一
个连接建立

处理 ServerSocket serverSocket = new ServerSocket(portNumber); ←
循环 Socket clientSocket = serverSocket.accept(); ←
开始 BufferedReader in = new BufferedReader(
3 new InputStreamReader(clientSocket.getInputStream()));
PrintWriter out =
    new PrintWriter(clientSocket.getOutputStream(), true); ←
String request, response;
4 while ((request = in.readLine()) != null) {
    if ("Done".equals(request)) {
        break;
    }
    response = processRequest(request);
    out.println(response);
}
继续执行处理循环
    这些流对象都派生于
    该套接字的流对象
    如果客户端发送了“Done”，
    则退出处理循环
    请求被传递给服
    务器的处理方法
    服务器的响应被
    发送给了客户端
    
```

代码清单 1-1 实现了 Socket API 的基本模式之一。以下是最重要的几点。

- ServerSocket 上的 accept() 方法将会一直阻塞到一个连接建立①，随后返回一个新的 Socket 用于客户端和服务器之间的通信。该 ServerSocket 将继续监听传入的连接。
- BufferedReader 和 PrintWriter 都衍生自 Socket 的输入输出流②。前者从一个字符输入流中读取文本，后者打印对象的格式化的表示到文本输出流。
- readLine() 方法将会阻塞，直到在③处一个由换行符或者回车符结尾的字符串被读取。
- 客户端的请求已经被处理④。

这段代码片段将只能同时处理一个连接，要管理多个并发客户端，需要为每个新的客户端 Socket 创建一个新的 Thread，如图 1-1 所示。

让我们考虑一下这种方案的影响。第一，在任何时候都可能有大量的线程处于休眠状态，只是等待输入或者输出数据就绪，这可能算是一种资源浪费。第二，需要为每个线程的调用栈都分配内存，其默认值大小区间为 64 KB 到 1 MB，具体取决于操作系统。第三，即使 Java 虚拟机（JVM）在物理上可以支持非常大数量的线程，但是远在到达该极限之前，上下文切换所带来的开销就会带来麻烦，例如，在达到 10 000 个连接的时候。

虽然这种并发方案对于支撑中小数量的客户端来说还算可以接受，但是为了支撑 100 000 或者更多的并发连接所需要的资源使得它很不理想。幸运的是，还有一种方案。

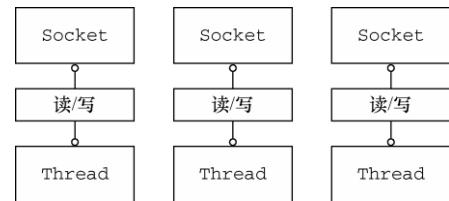


图 1-1 使用阻塞 I/O 处理多个连接

### 1.1.1 Java NIO

除了代码清单 1-1 中代码底层的阻塞系统调用之外，本地套接字库很早就提供了非阻塞调用，其为网络资源的利用率提供了相当多的控制：

*Text*

- 可以使用 setsockopt() 方法配置套接字，以便读/写调用在没有数据的时候立即返回，也就是说，如果是一个阻塞调用应该已经被阻塞了<sup>①</sup>；
- 可以使用操作系统的事件通知 API<sup>②</sup>注册一组非阻塞套接字，以确定它们中是否有任何的套接字已经有数据可供读写。

Java 对于非阻塞 I/O 的支持是在 2002 年引入的，位于 JDK 1.4 的 java.nio 包中。

① W. Richard Stevens 的 *Advanced Programming in the UNIX Environment* (Addison-Wesley, 1992) 第 364 页 “4.3BSD returned EWOULDBLOCK if an operation on a non-blocking descriptor could not complete without blocking”。

② 也称为 I/O 多路复用，该接口从最初的 select() 和 poll() 调用到更加高性能的实现，已经演变成了很多年。参见 Sangjin Han 的文章《Scalable Event Multiplexing: epoll vs. kqueue》( [www.eecs.berkeley.edu/~sangjin/2012/12/21 epoll-vs-kqueue.html](http://www.eecs.berkeley.edu/~sangjin/2012/12/21 epoll-vs-kqueue.html) )。

### 新的还是非阻塞的

NIO 最开始是新的输入/输出 ( New Input/Output ) 的英文缩写,但是,该 Java API 已经出现足够长的时间了,不再是“新的”了,因此,如今大多数的用户认为 NIO 代表非阻塞 I/O( Non-blocking I/O ),而阻塞 I/O( blocking I/O ) 是旧的输入/输出 ( old input/output, OIO )。你也可能遇到它被称为普通 I/O ( plain I/O ) 的时候。

## 1.1.2 选择器

图 1-2 展示了一个非阻塞设计,其实际上消除了上一节中所描述的那些弊端。

`class java.nio.channels.Selector` 是 Java 的非阻塞 I/O 实现的关键。它使用了事件通知 API 以确定在一组非阻塞套接字中有哪些已经就绪能够进行 I/O 相关的操作。因为可以在任何的时间检查任意的读操作或者写操作的完成状态,所以如图 1-2 所示,一个单一的线程便可以处理多个并发的连接。

总体来看,与阻塞 I/O 模型相比,这种模型提供了更好的资源管理:

- 使用较少的线程便可以处理许多连接,因此也减少了内存管理和上下文切换所带来的开销;
- 当没有 I/O 操作需要处理的时候,线程也可以被用于其他任务。

尽管已经有许多直接使用 Java NIO API 的应用程序被构建了,但是要做到如此正确和安全不容易。特别是,在高负载下可靠和高效地处理和调度 I/O 操作是一项繁琐而且容易出错的任务,最好留给高性能的网络编程专家——Netty。

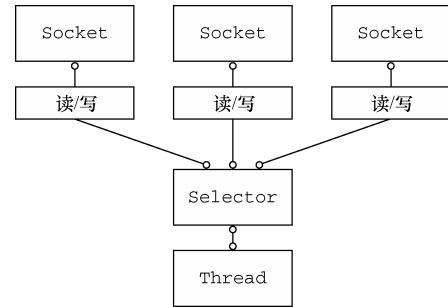


图 1-2 使用 Selector 的非阻塞 I/O

## 1.2 Netty 简介

不久以前,我们在本章一开始所呈现的场景——支持成千上万的并发客户端——还被认定为是不可能的。然而今天,作为系统用户,我们将这种能力视为理所当然;同时作为开发人员,我们期望将水平线提得更高<sup>①</sup>。因为我们知道,总会有更高的吞吐量和可扩展性的要求——在更低的成本的基础上进行交付。

不要低估了这最后一点的重要性。我们已经从漫长的痛苦经历中学到:直接使用底层的 API 暴露了复杂性,并且引入了对往往供不应求的技能的关键性依赖<sup>②</sup>。这也就是,面向对象的基本概念:用较简单的抽象隐藏底层实现的复杂性。

这一原则也催生了大量框架的开发,它们为常见的编程任务封装了解决方案,其中的许多都

① 这里指支撑更多的并发的客户端。——译者注

② 这里指熟悉这些底层的 API 的人员少。——译者注

和分布式系统的开发密切相关。我们可以确定地说：所有专业的Java开发人员都至少对它们熟知一二。<sup>①</sup>对于我们许多人来说，它们已经变得不可或缺，因为它们既能满足我们的技术需求，又能满足我们的时间表。

在网络编程领域，Netty是Java的卓越框架。<sup>②</sup>它驾驭了Java高级API的能力，并将其隐藏在一个易于使用的API之后。Netty使你可以专注于自己真正感兴趣的——你的应用程序的独一无二的价值。

在我们开始首次深入地了解 Netty 之前，请仔细审视表 1-1 中所总结的关键特性。有些是技术性的，而其他的更多的则是关于架构或设计哲学的。在本书的学习过程中，我们将不止一次地重新审视它们。

表 1-1 Netty 的特性总结

分 类	Netty 的特性
设计	统一的 API，支持多种传输类型，阻塞的和非阻塞的 <u>简单而强大的线程模型</u> 真正的无连接数据报套接字支持 链接逻辑组件以支持复用
易于使用	详实的Javadoc和大量的示例集 不需要超过JDK 1.6+ <sup>③</sup> 的依赖。(一些可选的特性可能需要Java 1.7+和/或额外的依赖)
性能	拥有比 Java 的核心 API 更高的吞吐量以及更低的延迟 <u>得益于池化和复用</u> ，拥有更低的资源消耗 最少的内存复制
健壮性	不会因为慢速、快速或者超载的连接而导致 OutOfMemoryError 消除在高速网络中 NIO 应用程序常见的不公平读/写比率
安全性	完整的 SSL/TLS 以及 StartTLS 支持 可用于受限环境下，如 Applet 和 OSGI
社区驱动	发布快速而且频繁

## 1.2.1 谁在使用 Netty

Netty拥有一个充满活力并且不断壮大的用户社区，其中不乏大型公司，如Apple、Twitter、Facebook、Google、Square和Instagram，还有流行的开源项目，如Infinispan、HornetQ、Vert.x、Apache Cassandra和Elasticsearch<sup>④</sup>，它们所有的核心代码都利用了Netty强大的网络抽象<sup>⑤</sup>。在初

<sup>①</sup> Spring 框架大概是最出名的，并且实际上是一个完整的应用程序框架的生态系统，处理了对象的创建、批量处理、数据库编程等。

<sup>②</sup> Netty 在 2011 年荣获了 Duke's Choice Award 的殊荣，参见 [www.java.net/dukeschoice/2011](http://www.java.net/dukeschoice/2011)。

<sup>③</sup> 最新的版本编译需要 JDK 1.8+，参见 <https://github.com/netty/netty/pull/6392>。——译者注

<sup>④</sup> 还包括炙手可热的大数据处理引擎 Spark。——译者注

<sup>⑤</sup> 完整的已知采用者列表参见 <http://netty.io/wiki/adopters.html>。

创企业中, Firebase 和 Urban Airship 也在使用 Netty, 前者用来自做 HTTP 长连接, 而后者用来支持各种各样的推送通知。

每当你使用 Twitter, 你便是在使用 Finagle<sup>①</sup>, 它们基于 Netty 的系统间通信框架。Facebook 在 Nifty 中使用了 Netty, 它们的 Apache Thrift 服务。可伸缩性和性能对这两家公司来说至关重要, 他们也经常为 Netty 贡献代码<sup>②</sup>。

反过来, Netty 也已从这些项目中受益, 通过实现 FTP、SMTP、HTTP 和 WebSocket 以及其他基于二进制和基于文本的协议, Netty 扩展了它的应用范围及灵活性。

## 1.2.2 异步和事件驱动

因为我们要大量地使用“异步”这个词, 所以现在是一个澄清上下文的好时机。异步(也就是非同步)事件肯定大家都熟悉。考虑一下电子邮件: 你可能会也可能不会收到你已经发出去的电子邮件对应的回复, 或者你也可能会在正在发送一封电子邮件的时候收到一个意外的消息。异步事件也可以具有某种有序的关系。通常, 你只有在已经问了一个问题之后才会得到一个和它对应的答案, 而在你等待它的同时你也可以做点别的事情。

在日常的生活中, 异步自然而然地就发生了, 所以你可能没有对它考虑过多少。但是让一个计算机程序以相同的方式工作就会产生一些非常特殊的问题。本质上, 一个既是异步的又是事件驱动的系统会表现出一种特殊的、对我们来说极具价值的行为: 它可以以任意的顺序响应在任意的时间点产生的事件。

这种能力对于实现最高级别的可伸缩性至关重要, 定义为: “一种系统、网络或者进程在需要处理的工作不断增长时, 可以通过某种可行的方式或者扩大它的处理能力来适应这种增长的能力。”<sup>③</sup>

异步和可伸缩性之间的联系又是什么呢?

- 非阻塞网络调用使得我们可以不必等待一个操作的完成。完全异步的 I/O 正是基于这个特性构建的, 并且更进一步: 异步方法会立即返回, 并且在它完成时, 会直接或者在稍后的某个时间点通知用户。
- 选择器使得我们能够通过较少的线程便可监视许多连接上的事件。

将这些元素结合在一起, 与使用阻塞 I/O 来处理大量事件相比, 使用非阻塞 I/O 来处理更快速、更经济。从网络编程的角度来看, 这是构建我们理想系统的关键, 而且你会看到, 这也是 Netty 的设计底蕴的关键。

<sup>①</sup> 关于 Finagle 的更多信息参见 <https://twitter.github.io/finagle/>。

<sup>②</sup> 第 15 章和第 16 章的案例研究描述了这里提到的公司中的一些是如何使用 Netty 来解决现实世界的问题的。

<sup>③</sup> André B. Bondi 的 *Proceedings of the second international workshop on Software and performance—WOSP'00* (2000) 第 195 页, “Characteristics of scalability and their impact on performance”。

在 1.3 节中，我们将首先看一看 Netty 的核心组件。现在，只需要将它们看作是域对象，而不是具体的 Java 类。随着时间的推移，我们将看到它们是如何协作，来为在网络上发生的事件提供通知，并使得它们可以被处理的。

## 1.3 Netty 的核心组件

在本节中我将要讨论 Netty 的主要构件块：

- Channel;
- 回调;
- Future;
- 事件和 ChannelHandler。

这些构建块代表了不同类型的构造：资源、逻辑以及通知。你的应用程序将使用它们来访问网络以及流经网络的数据。

对于每个组件来说，我们都将提供一个基本的定义，并且在适当的情况下，还会提供一个简单的示例代码来说明它的用法。

### 1.3.1 Channel

Channel 是 Java NIO 的一个基本构造。

它代表一个到实体（如一个硬件设备、一个文件、一个网络套接字或者一个能够执行一个或者多个不同的I/O操作的程序组件）的开放连接，如读操作和写操作<sup>①</sup>。

目前，可以把 Channel 看作是传入（入站）或者传出（出站）数据的载体。因此，它可以被打开或者被关闭，连接或者断开连接。

### 1.3.2 回调

一个回调其实就是一个方法，一个指向已经被提供给另外一个方法的方法的引用。这使得后者<sup>②</sup>可以在适当的时候调用前者。回调在广泛的编程场景中都有应用，而且也是在操作完成后通知相关方最常见的方式之一。

Netty 在内部使用了回调来处理事件；当一个回调被触发时，相关的事件可以被一个 `interface-ChannelHandler` 的实现处理。代码清单 1-2 展示了一个例子：当一个新的连接已经被建立时，`ChannelHandler` 的 `channelActive()` 回调方法将会被调用，并将打印出一条信息。

---

① Java 平台，标准版第 8 版 API 规范，`java.nio.channels`，`Channel`: <http://docs.oracle.com/javase/8/docs/api/java/nio/channels/package-summary.html>。

② 指接受回调的方法。——译者注

### 代码清单 1-2 被回调触发的 ChannelHandler

```
public class ConnectHandler extends ChannelInboundHandlerAdapter {
    @Override
    public void channelActive(ChannelHandlerContext ctx)
        throws Exception {
        System.out.println(
            "Client " + ctx.channel().remoteAddress() + " connected");
    }
}
```

当一个新的连接已经被建立时，  
channelActive(ChannelHandler  
Context)将会被调用

### 1.3.3 Future

Future 提供了另一种在操作完成时通知应用程序的方式。这个对象可以看作是一个异步操作的结果的占位符；它将在未来的某个时刻完成，并提供对其结果的访问。

JDK 预置了 interface `java.util.concurrent.Future`，但是其所提供的实现，只允许手动检查对应的操作是否已经完成，或者一直阻塞直到它完成。这是非常繁琐的，所以 Netty 提供了它自己的实现——`ChannelFuture`，用于在执行异步操作的时候使用。

`ChannelFuture`提供了几种额外的方法，这些方法使得我们能够注册一个或者多个 `ChannelFutureListener`实例。监听器的回调方法`operationComplete()`，将会在对应的操作完成时被调用<sup>①</sup>。然后监听器可以判断该操作是成功地完成了还是出错了。如果是后者，我们可以检索产生的`Throwable`。简而言之，由`ChannelFutureListener`提供的通知机制消除了手动检查对应的操作是否完成的必要。

每个 Netty 的出站 I/O 操作都将返回一个 `ChannelFuture`；也就是说，它们都不会阻塞。正如我们前面所提到过的一样，Netty 完全是异步和事件驱动的。

代码清单 1-3 展示了一个 `ChannelFuture` 作为一个 I/O 操作的一部分返回的例子。这里，`connect()`方法将会直接返回，而不会阻塞，该调用将会在后台完成。这究竟什么时候会发生则取决于若干的因素，但这个关注点已经从代码中抽象出来了。因为线程不用阻塞以等待对应的操作完成，所以它可以同时做其他的工作，从而更加有效地利用资源。

### 代码清单 1-3 异步地建立连接

```
Channel channel = ...;
// Does not block
ChannelFuture future = channel.connect(
    new InetSocketAddress("192.168.0.1", 25));
```

异步地连接到远程节点

<sup>①</sup> 如果在 `ChannelFutureListener` 添加到 `ChannelFuture` 的时候，`ChannelFuture` 已经完成，那么该 `ChannelFutureListener` 将会被直接地通知。——译者注

代码清单 1-4 显示了如何利用 ChannelFutureListener。首先，要连接到远程节点上。然后，要注册一个新的 ChannelFutureListener 到对 connect() 方法的调用所返回的 ChannelFuture 上。**当该监听器被通知连接已经建立的时候，要检查对应的状态①。**如果该操作是成功的，那么将数据写到该 Channel。否则，要从 ChannelFuture 中检索对应的数据。

#### 代码清单 1-4 回调实战

```

Channel channel = ...;
// Does not block
ChannelFuture future = channel.connect(    ← 异步地连接
    new InetSocketAddress("192.168.0.1", 25)); ← 到远程节点
future.addListener(new ChannelFutureListener() { ← 注册一个 ChannelFutureListener,
    @Override                                         以便在操作完成时获得通知
    public void operationComplete(ChannelFuture future) {
        ← 检查操作
        ← 的状态
        ① if (future.isSuccess()) { ← 如果操作是成功的，则创建
            ByteBuf buffer = Unpooled.copiedBuffer( ← 一个 ByteBuf 以持有数据
                "Hello", Charset.defaultCharset());
            ChannelFuture wf = future.channel()
                .writeAndFlush(buffer); ← 将数据异步地发送到远程节点。
                ....
        } else { ← 返回一个 ChannelFuture
            Throwable cause = future.cause(); ← 如果发生错误，则访问描述原因
            cause.printStackTrace();           ← 的 Throwable
        }
    });
}
);

```

需要注意的是，对错误的处理完全取决于你、目标，当然也包括目前任何对于特定类型的错误加以的限制。例如，如果连接失败，你可以尝试重新连接或者建立一个到另一个远程节点的连接。如果你把 ChannelFutureListener 看作是回调的一个更加精细的版本，那么你是对的。事实上，**回调和 Future 是相互补充的机制**；它们相互结合，构成了 Netty 本身的关键构件块之一。

#### 1.3.4 事件和 ChannelHandler

Netty 使用不同的事件来通知我们状态的改变或者是操作的状态。这使得我们能够基于已经发生的事件来触发适当的动作。这些动作可能是：

- 记录日志；
- **数据转换**；
- 流控制；
- 应用程序逻辑。

Netty 是一个网络编程框架，所以事件是按照它们与入站或出站数据流的相关性进行分类的。可能由入站数据或者相关状态更改而触发的事件包括：

- 连接已被激活或者连接失活；

- 数据读取；
- 用户事件；
- 错误事件。

出站事件是未来将会触发的某个动作的操作结果，这些动作包括：

- 打开或者关闭到远程节点的连接；
- 将数据写到或者冲刷到套接字。

每个事件都可以被分发给 ChannelHandler 类中的某个用户实现的方法。这是一个很好的将事件驱动范式直接转换为应用程序构件块的例子。图 1-3 展示了一个事件是如何被一个这样的 ChannelHandler 链处理的。

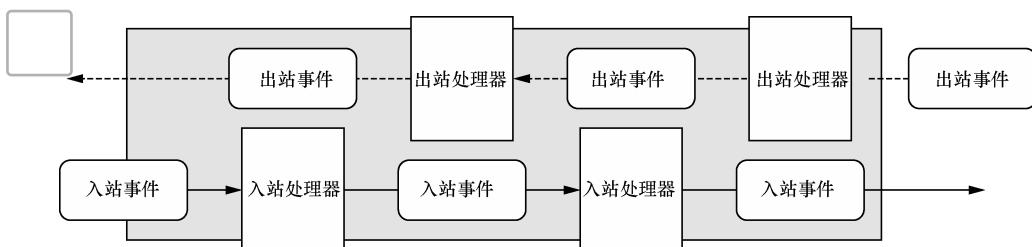


图 1-3 流经 ChannelHandler 链的入站事件和出站事件

Netty 的 ChannelHandler 为处理器提供了基本的抽象，如图 1-3 所示的那些。我们会在适当的时候对 ChannelHandler 进行更多的说明，但是目前你可以认为每个 ChannelHandler 的实例都类似于一种为了响应特定事件而被执行的回调。

Netty 提供了大量预定义的可以开箱即用的 ChannelHandler 实现，包括用于各种协议（如 HTTP 和 SSL/TLS）的 ChannelHandler。在内部，ChannelHandler 自己也使用了事件和 Future，使得它们也成为了你的应用程序将使用的相同抽象的消费者。

### 1.3.5 把它们放在一起

在本章中，我们介绍了 Netty 实现高性能网络编程的方式，以及它的实现中的一些主要的组件。让我们大体回顾一下我们讨论过的内容吧。

#### 1. Future、回调和 ChannelHandler

Netty 的异步编程模型是建立在 Future 和回调的概念之上的，而将事件派发到 ChannelHandler 的方法则发生在更深的层次上。结合在一起，这些元素就提供了一个处理环境，使你的应用程序逻辑可以独立于任何网络操作相关的顾虑而独立地演变。这也是 Netty 的设计方式的一个关键目标。

拦截操作以及高速地转换入站数据和出站数据，都只需要你提供回调或者利用操作所返回的 Future。这使得链接操作变得既简单又高效，并且促进了可重用的通用代码的编写。

## 2. 选择器、事件和 EventLoop

Netty 通过触发事件将 `Selector` 从应用程序中抽象出来，消除了所有本来将需要手动编写的派发代码。在内部，将会为每个 `Channel` 分配一个 `EventLoop`，用以处理所有事件，包括：

- 注册感兴趣的事件；
- 将事件派发给 `ChannelHandler`；
- 安排进一步的动作。

`EventLoop` 本身只由一个线程驱动，其处理了一个 `Channel` 的所有 I/O 事件，并且在该 `EventLoop` 的整个生命周期内都不会改变。这个简单而强大的设计消除了你可能有的在 `ChannelHandler` 实现中需要进行同步的任何顾虑，因此，你可以专注于提供正确的逻辑，用来在有兴趣的数据要处理的时候执行。如同我们在详细探讨 Netty 的线程模型时将会看到的，该 API 是简单而紧凑的。

## 1.4 小结

在这一章中，我们介绍了 Netty 框架的背景知识，包括 Java 网络编程 API 的演变过程，阻塞和非阻塞网络操作之间的区别，以及异步 I/O 在高容量、高性能的网络编程中的优势。

然后，我们概述了 Netty 的特性、设计和优点，其中包括 Netty 异步模型的底层机制，包括回调、`Future` 以及它们的结合使用。我们还谈到了事件是如何产生的以及如何拦截和处理它们。

在本书接下来的部分，我们将更加深入地探讨如何利用这些丰富的工具集来满足自己的应用程序的特定需求。

在下一章中，我们将要深入地探讨 Netty 的 API 以及编程模型的基础知识，而你则将编写你的第一款客户端和服务器应用程序。