# Essential Netty in Action 《Netty 实战(精髓)》

It is a book about the Essentials of Norman Maurer's [Netty in Action](http://manning.com/maurer/)(base on MEAP v10). Through this book, you can quickly start with Netty. This is a GitBook version of the book: <http://waylau.gitbooks.io/essential-netty-in-action/> Let's [READ](https://waylau.gitbooks.io/essential-netty-in-action/SUMMARY.md)!

《Netty 实战(精髓)》是对 Norman Maurer 的 《[Netty in Action](http://manning.com/maurer/)》(基于 MEAP v10)的一个中文精简。取其精华，去其糟粕，带你快速掌握 Netty ，插入配图，图文并茂方便用户理解。本书利用业余时间编写,由于时间紧凑,精力和能力有限,书中未免有纰漏和错误,望读者能够热忱斧正。

对于初学者，也推荐参阅《[Netty 4.x 用户指南](https://github.com/waylau/netty-4-user-guide)》。与之类似的 NIO 框架还有 MINA, 可参阅《[Apache MINA 2 用户指南](https://github.com/waylau/apache-mina-2.x-user-guide)》

## 关于开源

本项目是针对当前市面上缺乏 Netty 的相关的参考资料而产生的，广大 Netty 爱好者通过开源方式来学习交流 Netty，推动 [Netty 社区](http://netty.io/)的繁荣。

当然，广大开发者需认识到，开源不等于免费，对于原著的版权，仍应抱有敬意。请支持原著！

## 如何开始阅读

选择下面入口之一：

* <https://github.com/waylau/essential-netty-in-action/> 的 [SUMMARY.md](https://waylau.gitbooks.io/essential-netty-in-action/SUMMARY.md)（源码）
* <http://waylau.gitbooks.io/essential-netty-in-action/> 点击 Read 按钮（同步更新，国内访问速度一般）
* <https://waylau.com/essential-netty-in-action/>（国内访问速度快，定期更新。最后更新于 2016-2-16）

## 意见、建议

如有勘误、意见或建议欢迎拍砖 <https://github.com/waylau/essential-netty-in-action/issues>

## 联系作者:

您也可以直接联系我：

* 博客：[https://waylau.com](https://waylau.com/)
* 邮箱：[waylau521(at)gmail.com](mailto:waylau521@gmail.com)
* 微博：<http://weibo.com/waylau521>
* 开源：<https://github.com/waylau>

## 其他书籍

若您对本书不感冒，笔者还写了其他方面的超过一打的书籍（可见<https://waylau.com/books/>），多是开源电子书。

本人也维护了一个[books-collection](https://github.com/waylau/books-collection)项目，里面提供了优质的专门给程序员的开源、免费图书集合。

## 开源捐赠

捐赠所得所有款项将用于开源事业！

**开始**

# Netty-异步和数据驱动

## 什么是 Netty

Netty 是一个利用 Java 的高级网络的能力，隐藏其背后的复杂性而提供一个易于使用的 API 的客户端/服务器框架。Netty 提供高性能和可扩展性，让你可以自由地专注于你真正感兴趣的东西，你的独特的应用！

在这一章我们将解释 Netty 在处理一些高并发的网络问题体现的价值。然后，我们将介绍基本概念和构成 Netty 的工具包，我们将在这本书的其余部分深入研究。

## 一些历史

在网络发展初期，需要花很多时间来学习 socket 的复杂，寻址等等，在 C socket 库上进行编码，并需要在不同的操作系统上做不同的处理。

Java 早期版本(1995-2002)介绍了足够的面向对象的糖衣来隐藏一些复杂性，但实现复杂的客户端-服务器协议仍然需要大量的样板代码（和进行大量的监视才能确保他们是对的）。

这些早期的 Java API（java.net）只能通过原生的 socket 库来支持所谓的“blocking（阻塞）”的功能。一个简单的例子

Listing 1.1 Blocking I/O Example

ServerSocket serverSocket = new ServerSocket(portNumber);//1

Socket clientSocket = serverSocket.accept(); //2

BufferedReader in = new BufferedReader( //3

new InputStreamReader(clientSocket.getInputStream()));

PrintWriter out =

new PrintWriter(clientSocket.getOutputStream(), true);

String request, response;

while ((request = in.readLine()) != null) { //4

if ("Done".equals(request)) { //5

break;

}

}

response = processRequest(request); //6

out.println(response); //7

} //8

1.ServerSocket 创建并监听端口的连接请求

2.accept() 调用阻塞，直到一个连接被建立了。返回一个新的 Socket 用来处理 客户端和服务端的交互

3.流被创建用于处理 socket 的输入和输出数据。BufferedReader 读取从字符输入流里面的本文。PrintWriter 打印格式化展示的对象读到本文输出流

4.处理循环开始 readLine() 阻塞，读取字符串直到最后是换行或者输入终止。

5.如果客户端发送的是“Done”处理循环退出

6.执行方法处理请求，返回服务器的响应

7.响应发回客户端

8.处理循环继续

显然，这段代码限制每次只能处理一个连接。为了实现多个并行的客户端我们需要分配一个新的 Thread 给每个新的客户端 Socket(当然需要更多的代码)。但考虑使用这种方法来支持大量的同步，长连接。在任何时间点多线程可能处于休眠状态，等待输入或输出数据。这很容易使得资源的大量浪费，对性能产生负面影响。当然，有一种替代方案。

除了示例中所示阻塞调用，原生 socket 库同时也包含了非阻塞 I/O 的功能。这使我们能够确定任何一个 socket 中是否有数据准备读或写。我们还可以设置标志，因为读/写调用如果没有数据立即返回；就是说，如果一个阻塞被调用后就会一直阻塞，直到处理完成。通过这种方法，会带来更大的代码的复杂性成本，其实我们可以获得更多的控制权来如何利用网络资源。

## JAVA NIO

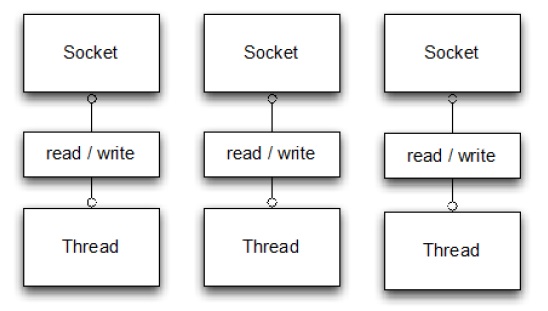
在 2002 年，Java 1.4 引入了非阻塞 API 在 java.nio 包（NIO）。

"New"还是"Nonblocking"?

NIO 最初是为 New Input/Output 的缩写。然而，Java 的 API 已经存在足够长的时间，它不再是新的。现在普遍使用的缩写来表示Nonblocking I/O (非阻塞 I/O)。另一方面，一般（包括作者）指阻塞 I/O 为 OIO 或 Old Input/Output。你也可能会遇到普通 I/O。

我们已经展示了在 Java 的 I/O 阻塞一例例子。图 1.1 展示了方法 必须扩大到处理多个连接：给每个连接创建一个线程，有些连接是空闲的！显然，这种方法的可扩展性将是受限于可以在 JVM 中创建的线程数。

Figure 1.1 Blocking I/O

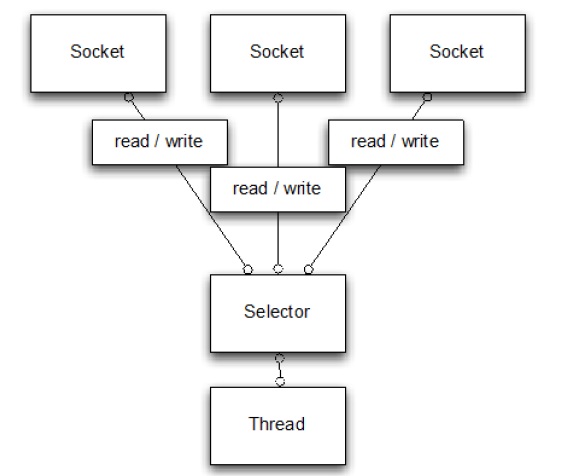


当你的应用中连接数比较少，这个方案还是可以接受。当并发连接超过10000 时，context-switching（上下文切换）开销将是明显的。此外，每个线程都有一个默认的堆栈内存分配了 128K 和 1M 之间的空间。考虑到整体的内存和操作系统需要处理 100000 个或更多的并发连接资源，这似乎是一个不理想的解决方案。

## SELECTOR

相比之下，图1.2 显示了使用非阻塞I/O，主要是消除了这些方法 约束。在这里，我们介绍了“Selector”，这是 Java 的无阻塞 I/O 实现的关键。

Figure 1.2 Nonblocking I/O



Selector 最终决定哪一组注册的 socket 准备执行 I/O。正如我们之前所解释的那样，这 I/O 操作设置为非阻塞模式。通过通知，一个线程可以同时处理多个并发连接。（一个 Selector 通常由一个线程处理，但具体实施可以使用多个线程。）因此，每次读或写操作执行能立即检查完成。总体而言，该模型提供了比 阻塞 I/O 模型 更好的资源使用，因为

* 可以用较少的线程处理更多连接，这意味着更少的开销在内存和上下文切换上
* 当没有 I/O 处理时，线程可以被重定向到其他任务上。

你可以直接用这些 Java API 构建的 NIO 建立你的应用程序，但这样做 正确和安全是无法保证的。实现可靠和可扩展的 event-processing（事件处理器）来处理和调度数据并保证尽可能有效地，这是一个繁琐和容易出错的任务，最好留给专家 - Netty。

Netty 介绍

一个应用想要支持成千上万并发的客户端，在以前，这样的想法会被认为是荒谬。而在今天，我们认为这是理所当然的。事实上，开发者知道，总是会有这样的需求——以较低的成本交付来换取更大的吞吐量和可用性。

我们不要低估最后一点的重要性。我们从漫长的痛苦的经验学习到，低级别的 API 不仅暴露了高级别直接使用的复杂性，而且引入了过分依赖于这项技术所造成的短板。因此，面向对象的一个基本原则：通过抽象来隐藏背后的复杂性。

这一原则已见成效，框架的形式封装解决方案成为了常见的编程任务，他们中有许多典型的分布式系统。现在大多数专业的 Java 开发人员都熟悉一个或多个这些框架（比如 Spring），并且许多已成为不可或缺的，使他们能够满足他们的技术要求以及他们的计划。

## 谁在用 Netty

Netty 是一个广泛使用的 Java 网络编程框架（Netty 在 2011 年获得了Duke's Choice Award，见<https://www.java.net/dukeschoice/2011>）。它活跃和成长于用户社区，像大型公司 Facebook 和 Instagram 以及流行 开源项目如 Infinispan, HornetQ, Vert.x, Apache Cassandra 和 Elasticsearch 等，都利用其强大的对于网络抽象的核心代码。

反过来，Netty 也从这些开源项目中获益。随着这些项目的作用，Netty 也不断提高了其应用的范围和灵活性，比如已经实现了的协议就有 FTP, SMTP, HTTP, WebSocket 和 SPDY 以及其他二进制和基于文本的协议。

在初创公司中 Firebase 和 Urban Airship 在使用 Netty。前者 Firebase 是使用 long-lived HTTP 连接，后者是使用 各种推送通知。

当你使用 Twitter,你会使用 Finagle,这个是基于 Netty API 提供给内部系统通讯。Facebook 使用 Netty 来提供于 Nifty 类似的功能 Apache Thrift 服务。这些公司可扩展性和高性能的表现得益于 Netty 的贡献。

这些例子的真实案例会在后面几章讲到。

2011 年 Netty 项目从 Red Hat 独立开来从而让广泛的开发者社区贡献者参与进来。Red Hat ，Twitter 继续使用 Netty ,并且成为保持其最活跃的贡献者之一。

下面展示了 Netty 技术和方法的特点

* 设计
  + 针对多种传输类型的统一接口 - 阻塞和非阻塞
  + 简单但更强大的线程模型
  + 真正的无连接的数据报套接字支持
  + 链接逻辑支持复用
* 易用性
  + 大量的 Javadoc 和 代码实例
  + 除了在 JDK 1.6 + 额外的限制。（一些特征是只支持在Java 1.7 +。可选的功能可能有额外的限制。）
* 性能
  + 比核心 Java API 更好的吞吐量，较低的延时
  + 资源消耗更少，这个得益于共享池和重用
  + 减少内存拷贝
* 健壮性
  + 消除由于慢，快，或重载连接产生的 OutOfMemoryError
  + 消除经常发现NIO 在高速网络中的应用中的不公平的读/写比
* 安全
  + 完整的 SSL / TLS 和 StartTLS 的支持
  + 运行在受限的环境例如 Applet 或 OSGI
* 社区
  + 发布的更早和更频繁
  + 社区驱动

## 异步和事件驱动

所有的网络应用程序需要被设计为可扩展性，可以被界定为“一个系统，网络能力，或过程中能够处理越来越多的工作方式或可扩大到容纳增长的能力”（见 Bondi, André B. (2000). "Characteristics of scalability and their impact on performance"）。我们已经说过，Netty 帮助您利用非阻塞 I/O 完成这一目标，通常称为“异步 I/O”

我们将使用“异步”和其同源词在这本书中大量的使用，所以这是介绍他们的一个很好的时候。异步，即非同步事件，当然是跟你日常生活的类似。例如，您可以发送电子邮件；可能得到或者得不到任何回应，或者当你发送一个您可能会收到一个消息。异步事件也可以有一个有序的关系。例如，你通常不会收到一个问题的答案直到提出一个问题，但是你并没有阻止同时一些其他的东西。

在日常生活中异步就这样发生了，所以我们不会经常想到。但让计算机程序的工作方式，来实现我们提出了的特殊的问题，会有一点复杂。在本质上，一个系统是异步和“事件驱动”将会表现出一个特定的，对我们来说，有价值的 行为：它可以响应在任何时间以任何顺序发生的事件。

这是我们要建立一种制度，正如我们将会看到，这是典范的 Netty 自底向上的支持。

# 构成部分

正如我们前面解释的，非阻塞 I/O 不会强迫我们等待操作的完成。在这种能力的基础上，真正的异步 I/O 起到了更进一步的作用:一个异步方法完成时立即返回并直接或稍后通知用户。

正如我们将看到的,在一个网络环境的异步模型可以更有效地利用资源,可以快速连续执行多个调用。

### Channel

[Channel](http://docs.oracle.com/javase/7/docs/api/java/nio/channels/Channel.html) 是 NIO 基本的结构。它代表了一个用于连接到实体如硬件设备、文件、网络套接字或程序组件,能够执行一个或多个不同的 I/O 操作（例如读或写）的开放连接。

现在,把 Channel 想象成一个可以“打开”或“关闭”,“连接”或“断开”和作为传入和传出数据的运输工具。

### Callback (回调)

callback (回调)是一个简单的方法,提供给另一种方法作为引用,这样后者就可以在某个合适的时间调用前者。这种技术被广泛使用在各种编程的情况下,最常见的方法之一通知给其他人操作已完成。

Netty 内部使用回调处理事件时。一旦这样的回调被触发，事件可以由接口 ChannelHandler 的实现来处理。如下面的代码，一旦一个新的连接建立了,调用 channelActive(),并将打印一条消息。

Listing 1.2 ChannelHandler triggered by a callback

public class ConnectHandler extends ChannelInboundHandlerAdapter {

@Override

public void channelActive(ChannelHandlerContext ctx) throws Exception { //1

System.out.println(

"Client " + ctx.channel().remoteAddress() + " connected");

}

}

1.当建立一个新的连接时调用 ChannelActive()

### Future

Future 提供了另外一种通知应用操作已经完成的方式。这个对象作为一个异步操作结果的占位符,它将在将来的某个时候完成并提供结果（me：对象保存的是对堆空间的引用，所以可以在未来的时候在其他的一个引用改变对象的时候该对象的状态能够被其他的引用查看到）。

JDK 附带接口 java.util.concurrent.Future ,但所提供的实现只允许您手动检查操作是否完成或阻塞了。这是很麻烦的，所以 Netty 提供自己的实现,ChannelFuture,用于在执行异步操作时使用。

ChannelFuture 提供多个附加方法来允许一个或者多个 ChannelFutureListener 实例。这个回调方法 operationComplete() 会在操作完成时调用。事件监听者能够确认这个操作是否成功或者是错误。如果是后者,我们可以检索到产生的 Throwable。简而言之, ChannelFutureListener 提供的通知机制（me：js中有很多的这样监听）不需要手动检查操作是否完成的。

每个 Netty 的 outbound I/O 操作都会返回一个 ChannelFuture;这样就不会阻塞。这就是 Netty 所谓的“自底向上的异步和事件驱动”。

下面例子简单的演示了作为 I/O 操作的一部分 ChannelFuture 的返回。当调用 connect() 将会直接是非阻塞的，并且调用在背后完成。由于线程是非阻塞的，所以无需等待操作完成，而可以去干其他事，因此这使得资源利用更高效。

Listing 1.3 Callback in action

Channel channel = ...;

//不会阻塞

ChannelFuture future = channel.connect(

new InetSocketAddress("192.168.0.1", 25));//1

1.异步连接到远程地址

下面代码描述了如何利用 ChannelFutureListener 。首先，连接到远程地址。接着，通过 ChannelFuture 调用 connect() 来 注册一个新ChannelFutureListener。当监听器被通知连接完成，我们检查状态。如果是成功，就写数据到 Channel，否则我们检索 ChannelFuture 中的Throwable。

注意，错误的处理取决于你的项目。当然,特定的错误是需要加以约束 的。例如,在连接失败的情况下你可以尝试连接到另一个。

Listing 1.4 Callback in action

Channel channel = ...;

//不会阻塞

ChannelFuture future = channel.connect( //1

new InetSocketAddress("192.168.0.1", 25));

future.addListener(new ChannelFutureListener() { //2

@Override

public void operationComplete(ChannelFuture future) {

if (future.isSuccess()) { //3

ByteBuf buffer = Unpooled.copiedBuffer(

"Hello", Charset.defaultCharset()); //4

ChannelFuture wf = future.channel().writeAndFlush(buffer); //5

// ...

} else {

Throwable cause = future.cause(); //6

cause.printStackTrace();

}

}

});

1.异步连接到远程对等节点。调用立即返回并提供 ChannelFuture。

2.操作完成后通知注册一个 ChannelFutureListener 。

3.当 operationComplete() 调用时检查操作的状态。

4.如果成功就创建一个 ByteBuf 来保存数据。

5.异步发送数据到远程。再次返回ChannelFuture。

6.如果有一个错误则抛出 Throwable,描述错误原因。

### Event 和 Handler

Netty 使用不同的事件来通知我们更改的状态或操作的状态。这使我们能够根据发生的事件触发适当的行为。

这些行为可能包括：

* 日志
* 数据转换
* 流控制
* 应用程序逻辑

由于 Netty 是一个网络框架,事件很清晰的跟入站或出站数据流相关。因为一些事件可能触发传入的数据或状态的变化包括:

* 活动或非活动连接
* 数据的读取
* 用户事件
* 错误

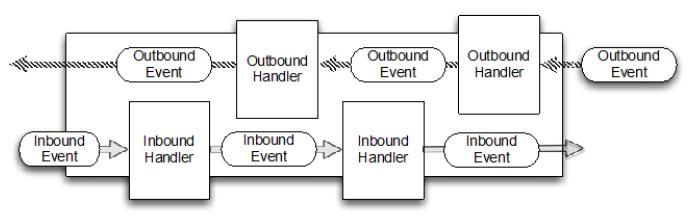
出站事件是由于在未来操作将触发一个动作。这些包括:

* 打开或关闭一个连接到远程
* 写或冲刷数据到 socket

每个事件都可以分配给用户实现处理程序类的方法。这说明了事件驱动的范例可直接转换为应用程序构建块。

图1.3显示了一个事件可以由一连串的事件处理器来处理

Figure 1.3 Event Flow



Netty 的 ChannelHandler 是各种处理程序的基本抽象。想象下，每个处理器实例就是一个回调，用于执行对各种事件的响应。

在此基础之上，Netty 也提供了一组丰富的预定义的处理程序,您可以开箱即用。比如，各种协议的编解码器包括 HTTP 和 SSL/TLS。在内部,ChannelHandler 使用事件和 future 本身,创建具有 Netty 特性抽象的消费者。

### 整合

#### FUTURE, CALLBACK 和 HANDLER

Netty 的异步编程模型是建立在 future 和 callback 的概念上的。所有这些元素的协同为自己的设计提供了强大的力量。

拦截操作和转换入站或出站数据只需要您提供回调或利用 future 操作返回的。这使得操作简单、高效,促进编写可重用的、通用的代码。Netty 设计的一个主要目标是促进“关注点分离”:你的业务逻辑从网络基础设施应用程序中分离。

#### SELECTOR, EVENT 和 EVENT LOOP

Netty 通过触发事件从应用程序中抽象出 Selector，从而避免手写调度代码。EventLoop 分配给每个 Channel 来处理所有的事件，包括

* 注册感兴趣的事件
* 调度事件到 ChannelHandler
* 安排进一步行动

该 EventLoop 本身是由只有一个线程驱动，它给一个 Channel 处理所有的 I/O 事件，并且在 EventLoop 的生命周期内不会改变。这个简单而强大的线程模型消除你可能对你的 ChannelHandler 同步的任何关注，这样你就可以专注于提供正确的回调逻辑来执行。该 API 是简单和紧凑。

# 关于本书

我们开始通过讨论阻塞和非阻塞处理之间的差异来了解到后一种方法的优点。然后，我们转移到了的 Netty的功能，设计和效益的概述。这些包括了Netty 的异步模型，包括回调，future 及其组合使用。我们还谈到了Netty 的线程模型，事件是如何被使用的，以及它们如何被拦截和处理。展望未来，我们将更加深入探索如何使用这些丰富的工具集用来满足特殊需求的应用。

一路上，我们将介绍公司的工程师自己的案例研究解释为什么他们选择的Netty 以及他们如何使用它。

因此，让我们开始吧。在下一章中，我们将深入研究了 Netty 的 API 的基础知识，编程模型，开始写 echo（回声）服务器和客户端。

# 第一个 Netty 应用

在本章中，首先你要确保你有一个可以工作的开发环境，并通过构建一个简单的客户端和服务器来进行测试。虽然在开始下一章节前，我们还不会开始学习Netty 框架的细节，但在这里我们将会仔细观察我们所引入的 API 方面的内容，即通过 ChannelHandler 来实现应用的逻辑。

**设置开发环境**

如果你已经有了 Maven 的开发环境，那你可以跳过本节。

本书例子需要 JDK 和 Apache Maven,都可以免费下载到。

**1.安装配置 JDK**

建议用 JDK 7+

**2.下载 IDE**

JAVA 的 IDE 很多，主流的有

* Eclipse: [http://www.eclipse.org](http://www.eclipse.org/)
* NetBeans: [http://www.netbeans.org](http://www.netbeans.org/)
* Intellij Idea Community Edition: [http://www.jetbrains.com](http://www.jetbrains.com/)

**3.下载安装 Maven**

可以参考：<http://www.waylau.com/apache-maven-3-1-0-installation-deployment-and-use/>

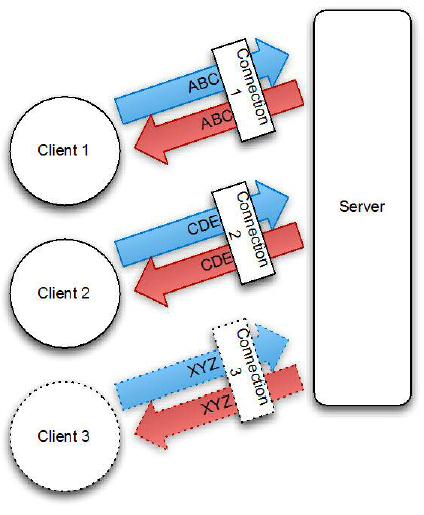
**4.配置工具**

确保系统环境变量有 JAVA\_HOME 和 M2\_HOME

# Netty 客户端/服务器 总览

在本节中，我们将构建一个完整的的 Netty客户端和服务器。虽然你可能集中在写客户端是浏览器的基于 Web 的服务，接下来你将会获得更完整了解 Netty 的 API 是如何实现客户端和服务器的。

Figure 2.1.Echo client / server



图中显示了连接到服务器的多个并发的客户端。在理论上，客户端可以支持的连接数只受限于使用的 JDK 版本中的制约。

echo（回声）客户端和服务器之间的交互是很简单的;客户端启动后，建立一个连接发送一个或多个消息发送到服务器，其中每相呼应消息返回给客户端。诚然，这个应用程序并不是非常有用。但这项工作是为了更好的理解请求 - 响应交互本身，这是一个基本的模式的客户端/服务器系统。

我们将通过检查服务器端代码开始。

**Netty 快速入门**

下面枚举所有的 Netty 应用程序的基本构建模块，包括客户端和服务器。

**BOOTSTRAP**

Netty 应用程序通过设置 bootstrap（引导）类的开始，该类提供了一个用于应用程序网络层配置的容器。

**CHANNEL**

底层网络传输 API 必须提供给应用I/O操作的接口，如读，写，连接，绑定等等。对于我们来说，这是结构几乎总是会成为一个“socket”。 Netty 中的接口 Channel 定义了与 socket 丰富交互的操作集：bind, close, config, connect, isActive, isOpen, isWritable, read, write 等等。 Netty 提供大量的 Channel 实现来专门使用。这些包括 AbstractChannel，AbstractNioByteChannel，AbstractNioChannel，EmbeddedChannel， LocalServerChannel，NioSocketChannel 等等。

**CHANNELHANDLER**

ChannelHandler 支持很多协议，并且提供用于数据处理的容器。我们已经知道 ChannelHandler 由特定事件触发。 ChannelHandler 可专用于几乎所有的动作，包括将一个对象转为字节（或相反），执行过程中抛出的异常处理。

常用的一个接口是 ChannelInboundHandler，这个类型接收到入站事件（包括接收到的数据）可以处理应用程序逻辑。当你需要提供响应时，你也可以从 ChannelInboundHandler 冲刷数据。一句话，业务逻辑经常存活于一个或者多个 ChannelInboundHandler。

**CHANNELPIPELINE**

ChannelPipeline 提供了一个容器给 ChannelHandler 链并提供了一个API 用于管理沿着链入站和出站事件的流动。每个 Channel 都有自己的ChannelPipeline，当 Channel 创建时自动创建的。 ChannelHandler 是如何安装在 ChannelPipeline？ 主要是实现了ChannelHandler 的抽象 ChannelInitializer。ChannelInitializer子类 通过 ServerBootstrap 进行注册。当它的方法 initChannel() 被调用时，这个对象将安装自定义的 ChannelHandler 集到 pipeline。当这个操作完成时，ChannelInitializer 子类则 从 ChannelPipeline 自动删除自身。

**EVENTLOOP**

EventLoop 用于处理 Channel 的 I/O 操作。一个单一的 EventLoop通常会处理多个 Channel 事件。一个 EventLoopGroup 可以含有多于一个的 EventLoop 和 提供了一种迭代用于检索清单中的下一个。

**CHANNELFUTURE**

Netty 所有的 I/O 操作都是异步。因为一个操作可能无法立即返回，我们需要有一种方法在以后确定它的结果。出于这个目的，Netty 提供了接口 ChannelFuture,它的 addListener 方法注册了一个 ChannelFutureListener ，当操作完成时，可以被通知（不管成功与否）。

*更多关于 ChannelFuture*

*想想一个 ChannelFuture 对象作为一个未来执行操作结果的占位符。何时执行取决于几个因素，因此不可能预测与精确。但我们可以肯定的是，它会被执行。此外，所有的操作返回 ChannelFuture 对象和属于同一个 Channel 将在以正确的顺序被执行，在他们被调用后。*