# 第02章-你的第一款Netty应用程序

在本章中，我们将展示如何构建一个基于Netty 的客户端和服务器。应用程序很简单：客户端将消息发送给服务器，而服务器再将消息回送给客户端。但是这个练习很重要，原因有两个。

首先，它会提供一个测试台，用于设置和验证你的开发工具和环境，如果你打算通过对本书的示例代码的练习来为自己将来的开发工作做准备，那么它将是必不可少的。

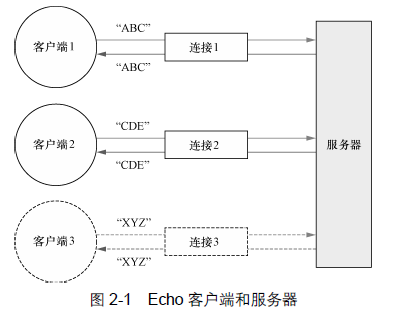
其次，你将获得关于Netty 的一个关键方面的实践经验，即在前一章中提到过的：通过ChannelHandler 来构建应用程序的逻辑。这能让你对Netty API 的深入学习做好准备。

## 2.1 设置开发环境

要编译和运行本书的示例，只需要JDK 和Apache Maven 这两样工具，它们都是可以免费下载的。可以使用纯文本编辑器，但是建议你使用用于Java 的集成开发环境（IDE）。

## 2.2 Netty 客户端/服务器概览

图2-1 从高层次上展示了一个你将要编写的Echo 客户端和服务器应用程序。虽然你的主要关注点可能是编写基于Web 的用于被浏览器访问的应用程序，但是通过同时实现客户端和服务器，你一定能更加全面地理解Netty 的API。



虽然我们已经谈及到了客户端，但是该图展示的是多个客户端同时连接到一台服务器。所能够支持的客户端数量，在理论上，仅受限于系统的可用资源（以及所使用的JDK 版本可能会施加的限制）。

Echo 客户端和服务器之间的交互是非常简单的；在客户端建立一个连接之后，它会向服务器发送一个或多个消息，反过来，服务器又会将每个消息回送给客户端。虽然它本身看起来好像用处不大，但它充分地体现了客户端/服务器系统中典型的请求-响应交互模式。

## 2.3 编写Echo 服务器

所有的Netty 服务器都需要以下两部分。

* 至少一个ChannelHandler—该组件实现了服务器对从客户端接收的数据的处理，即它的业务逻辑。
* 引导—这是配置服务器的启动代码。至少，它会将服务器绑定到它要监听连接请求的端口上。

在本小节的剩下部分，我们将描述Echo 服务器的业务逻辑以及引导代码。

### 2.3.1 ChannelHandler 和业务逻辑

在第1 章中，我们介绍了Future 和回调，并且阐述了它们在事件驱动设计中的应用。我们还讨论了ChannelHandler，它是一个接口族的父接口，它的实现负责接收并响应事件通知。在Netty 应用程序中，所有的数据处理逻辑都包含在这些核心抽象的实现中。

因为你的Echo 服务器会响应传入的消息，所以它需要实现ChannelInboundHandler 接口，用来定义响应入站事件的方法。这个简单的应用程序只需要用到少量的这些方法，所以继承Channel-InboundHandlerAdapter 类也就足够了，它提供了ChannelInboundHandler 的默认实现。

我们感兴趣的方法是：

* channelRead()——对于每个传入的消息都要调用；
* channelReadComplete()——通知ChannelInboundHandler 最后一次对channel-Read()的调用是当前批量读取中的最后一条消息；
* exceptionCaught()——在读取操作期间，有异常抛出时会调用。

该Echo 服务器的ChannelHandler 实现是EchoServerHandler，如代码清单2-1 所示。

|  |
| --- |
| *// 代码清单2-1 EchoServerHandler*  *// 标示一个ChannelHandler可以被多个Channel安全地共享* @ChannelHandler.Sharable **public class** EchoServerHandler **extends** ChannelInboundHandlerAdapter {  @Override  **public void** channelRead(ChannelHandlerContext ctx, Object msg) **throws** Exception {  *// 将消息记录到控制台* ByteBuf in = (ByteBuf) msg;  System.***out***.println( **"Server received: "** + in.toString(CharsetUtil.***UTF\_8***));  *// 将接收到的消息写给发送者，而不冲刷出站消息* ctx.write(in);  }   @Override  **public void** channelReadComplete(ChannelHandlerContext ctx) **throws** Exception {  *// 将未决消息冲刷到远程节点，并且关闭该Channel* ctx.writeAndFlush(Unpooled.***EMPTY\_BUFFER***).addListener(ChannelFutureListener.***CLOSE***);  }   @Override  **public void** exceptionCaught(ChannelHandlerContext ctx, Throwable cause) **throws** Exception {  *// 打印异常栈跟踪* cause.printStackTrace();  *// 关闭该Channel* ctx.close();  } } |

ChannelInboundHandlerAdapter 有一个直观的API，并且它的每个方法都可以被重写以挂钩到事件生命周期的恰当点上。因为需要处理所有接收到的数据，所以你重写了channelRead()方法。在这个服务器应用程序中，你将数据简单地回送给了远程节点。

重写exceptionCaught()方法允许你对Throwable 的任何子类型做出反应，在这里你记录了异常并关闭了连接。虽然一个更加完善的应用程序也许会尝试从异常中恢复，但在这个场景下，只是通过简单地关闭连接来通知远程节点发生了错误。

如果不捕获异常，会发生什么呢

每个Channel 都拥有一个与之相关联的ChannelPipeline，其持有一个ChannelHandler 的实例链。在默认的情况下，ChannelHandler 会把对它的方法的调用转发给链中的下一个Channel-Handler。因此，如果exceptionCaught()方法没有被该链中的某处实现，那么所接收的异常将会被传递到ChannelPipeline 的尾端并被记录。为此，你的应用程序应该提供至少有一个实现了exceptionCaught()方法的ChannelHandler。

除了ChannelInboundHandlerAdapter 之外，还有很多需要学习的ChannelHandler 的子类型和实现，下面这些是关键点：

* 针对不同类型的事件来调用ChannelHandler；
* 应用程序通过实现或者扩展ChannelHandler 来挂钩到事件的生命周期，并且提供自定义的应用程序逻辑；
* 在架构上，ChannelHandler 有助于保持业务逻辑与网络处理代码的分离。这简化了开发过程，因为代码必须不断地演化以响应不断变化的需求。