# 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。

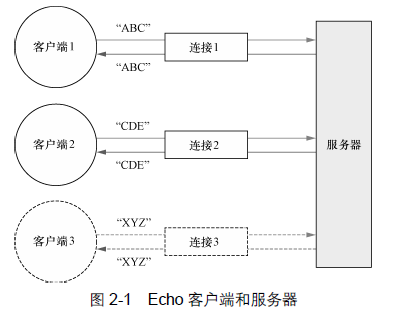
其次，你将获得关于Netty 的一个关键方面的实践经验，即在前一章中提到过的：通过ChannelHandler 来构建应用程序的逻辑。这能让你对Netty API 的深入学习做好准备。

## 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。

### 2.3.2 引导服务器

在讨论过由EchoServerHandler 实现的核心业务逻辑之后，我们现在可以探讨引导服务器本身的过程了，具体涉及以下内容：

* 绑定到服务器将在其上监听并接受传入连接请求的端口；
* 配置Channel，以将有关的入站消息通知给EchoServerHandler 实例。

传输

在这一节中，你将遇到术语传输。在网络协议的标准多层视图中，传输层提供了端到端的或者主机到主机的通信服务。因特网通信是建立在TCP 传输之上的。

除了一些由Java NIO 实现提供的服务器端性能增强之外，NIO 传输大多数时候指的就是TCP 传输。

代码清单2-2 展示了EchoServer 类的完整代码。

|  |
| --- |
| **// 代码清单2-2 EchoServer 类**  **public class** EchoServer {  **private final int port**;   **public** EchoServer(**int** port) {  **this**.**port** = port;  }   **public static void** main(String[] args) **throws** Exception {   **if** (args == **null** || args.**length** == **0**) {  args = **new** String[]{**"8888"**};  }   **if** (args.**length** != **1**) {  System.***err***.println(**"Usage: "** + EchoServer.**class**.getSimpleName() + **" <port>"**);  **return**;  }  *// 设置端口值（如果端口参数的格式不正确，则抛出一个NumberFormatException）* **int** port = Integer.*parseInt*(args[**0**]);  *// 调用服务器的start()方法* **new** EchoServer(port).start();  }   **private void** start() **throws** Exception {  **final** EchoServerHandler serverHandler = **new** EchoServerHandler();  *// 1、创建Event-LoopGroup* EventLoopGroup group = **new** NioEventLoopGroup();  **try** {  *// 2、创建Server-Bootstrap* ServerBootstrap b = **new** ServerBootstrap();  b.group(group)  .channel(NioServerSocketChannel.**class**) *// 3、指定所使用的NIO传输Channel* .localAddress(**new** InetSocketAddress(**port**)) *// 4、使用指定的端口设置套接字地址* .childHandler(**new** ChannelInitializer<SocketChannel>() { *// 5、添加一个EchoServer-Handler 到子Channel的ChannelPipeline* @Override  **protected void** initChannel(SocketChannel ch) **throws** Exception {  *// EchoServerHandler 被标注为@Shareable，所以我们可以总是使用同样的实例  // 这里对于所有的客户端连接来说，都会使用同一个EchoServerHandler，  // 因为其被标注为@Sharable，这将在后面的章节中讲到* ch.pipeline().addLast(serverHandler);  }  });  *// 6、异步地绑定服务器；调用sync()方法阻塞等待直到绑定完成* ChannelFuture f = b.bind().sync();  System.***out***.println(EchoServer.**class**.getName()  + **" started and listening for connections on "** + f.channel().localAddress());  *// 7、获取Channel 的CloseFuture，并且阻塞当前线程直到它完成* f.channel().closeFuture().sync();  } **finally** {  *// 8、关闭EventLoopGroup，释放所有的资源* group.shutdownGracefully().sync();  }  }  } |

在（2）处，你创建了一个ServerBootstrap 实例。因为你正在使用的是NIO 传输，所以你指定了NioEventLoopGroup（1） 来接受和处理新的连接，并且将Channel 的类型指定为NioServer-SocketChannel （3）。在此之后，你将本地地址设置为一个具有选定端口的InetSocket-Address （4）。服务器将绑定到这个地址以监听新的连接请求。

在（5）处，你使用了一个特殊的类——ChannelInitializer。这是关键。当一个新的连接被接受时，一个新的子Channel 将会被创建，而ChannelInitializer 将会把一个你的EchoServerHandler 的实例添加到该Channel 的ChannelPipeline 中。正如我们之前所解释的，这个ChannelHandler 将会收到有关入站消息的通知。

虽然NIO 是可伸缩的，但是其适当的尤其是关于多线程处理的配置并不简单。Netty 的设计封装了大部分的复杂性。

接下来你绑定了服务器（6），并等待绑定完成。（对sync()方法的调用将导致当前Thread阻塞，一直到绑定操作完成为止）。在处，该应用程序将会阻塞等待直到服务器的Channel关闭（因为你在Channel 的CloseFuture 上调用了sync()方法）。然后，你将可以关闭EventLoopGroup，并释放所有的资源，包括所有被创建的线程。

这个示例使用了NIO，因为得益于它的可扩展性和彻底的异步性，它是目前使用最广泛的传输。但是也可以使用一个不同的传输实现。如果你想要在自己的服务器中使用OIO 传输，将需要指定OioServerSocketChannel 和OioEventLoopGroup。

与此同时，让我们回顾一下你刚完成的服务器实现中的重要步骤。下面这些是服务器的主要代码组件：

* EchoServerHandler 实现了业务逻辑；
* main()方法引导了服务器；

引导过程中所需要的步骤如下：

* 创建一个ServerBootstrap 的实例以引导和绑定服务器；
* 创建并分配一个NioEventLoopGroup 实例以进行事件的处理，如接受新连接以及读/写数据；

指定服务器绑定的本地的InetSocketAddress；

* 使用一个EchoServerHandler 的实例初始化每一个新的Channel；
* 调用ServerBootstrap.bind()方法以绑定服务器。

## 2.4 编写Echo 客户端

Echo 客户端将会：

（1）连接到服务器；

（2）发送一个或者多个消息；

（3）对于每个消息，等待并接收从服务器发回的相同的消息；

（4）关闭连接。

编写客户端所涉及的两个主要代码部分也是业务逻辑和引导，和你在服务器中看到的一样。

### 2.4.1 通过ChannelHandler 实现客户端逻辑

如同服务器，客户端将拥有一个用来处理数据的ChannelInboundHandler。在这个场景下，你将扩展SimpleChannelInboundHandler 类以处理所有必须的任务，如代码清单2-3所示。这要求重写下面的方法：

* channelActive()——在到服务器的连接已经建立之后将被调用；
* channelRead0()——当从服务器接收到一条消息时被调用；
* exceptionCaught()——在处理过程中引发异常时被调用。

|  |
| --- |
| *// 代码清单2-3 客户端的ChannelHandler*  *// 标示一个ChannelHandler 可以被多个Channel 安全地共享* @ChannelHandler.Sharable **public class** EchoClientHandler **extends** SimpleChannelInboundHandler<ByteBuf> {   @Override  **public void** channelActive(ChannelHandlerContext ctx) **throws** Exception {  *// 当被通知Channel是活跃的时候，发送一条消息* ctx.writeAndFlush(Unpooled.*copiedBuffer*(**"Netty rocks!"**, CharsetUtil.***UTF\_8***));  }   @Override  **protected void** channelRead0(ChannelHandlerContext ctx, ByteBuf in) **throws** Exception {  *// 记录已接收消息的转储* System.***out***.println(**"Client received: "** + in.toString(CharsetUtil.***UTF\_8***));  }   @Override  **public void** exceptionCaught(ChannelHandlerContext ctx, Throwable cause) **throws** Exception {  *// 在发生异常时，记录错误并关闭Channel* cause.printStackTrace();  ctx.close();  } } |

首先，你重写了channelActive()方法，其将在一个连接建立时被调用。这确保了数据将会被尽可能快地写入服务器，其在这个场景下是一个编码了字符串"Netty rocks!"的字节缓冲区。

接下来，你重写了channelRead0()方法。每当接收数据时，都会调用这个方法。需要注意的是，由服务器发送的消息可能会被分块接收。也就是说，如果服务器发送了5 字节，那么不能保证这5 字节会被一次性接收。即使是对于这么少量的数据，channelRead0()方法也可能会被调用两次，第一次使用一个持有3 字节的ByteBuf（Netty 的字节容器），第二次使用一个持有2 字节的ByteBuf。作为一个面向流的协议，TCP 保证了字节数组将会按照服务器发送它们的顺序被接收。

重写的第三个方法是exceptionCaught()。如同在EchoServerHandler（见代码清单2-2）中所示，记录Throwable，关闭Channel，在这个场景下，终止到服务器的连接。

SimpleChannelInboundHandler 与ChannelInboundHandler

你可能会想：为什么我们在客户端使用的是SimpleChannelInboundHandler，而不是在Echo-ServerHandler 中所使用的ChannelInboundHandlerAdapter 呢？这和两个因素的相互作用有关：业务逻辑如何处理消息以及Netty 如何管理资源。

在客户端，当channelRead0()方法完成时，你已经有了传入消息，并且已经处理完它了。当该方法返回时，SimpleChannelInboundHandler 负责释放指向保存该消息的ByteBuf 的内存引用。

在EchoServerHandler 中，你仍然需要将传入消息回送给发送者，而write()操作是异步的，直到channelRead()方法返回后可能仍然没有完成（如代码清单2-1 所示）。为此，EchoServerHandler扩展了ChannelInboundHandlerAdapter，其在这个时间点上不会释放消息。

消息在EchoServerHandler 的channelReadComplete()方法中，当writeAndFlush()方法被调用时被释放（见代码清单2-1）。