# 【第04章-传输】

流经网络的数据总是具有相同的类型：字节。这些字节是如何流动的主要取决于我们所说的网络传输—一个帮助我们抽象底层数据传输机制的概念。用户并不关心这些细节；他们只想确保他们的字节被可靠地发送和接收。

Netty 为它所有的传输实现提供了一个通用API，这使得这种转换比你直接使用JDK所能够达到的简单得多。所产生的代码不会被实现的细节所污染，而你也不需要在你的整个代码库上进行广泛的重构。

## 4.1 案例研究：传输迁移

我们将从一个应用程序开始我们对传输的学习，这个应用程序只简单地接受连接，向客户端写“Hi!”，然后关闭连接。

### 4.1.1 不通过Netty 使用OIO 和NIO

我们将介绍仅使用了JDK API 的应用程序的阻塞（OIO）版本和异步（NIO）版本。代码清单4-1 展示了其阻塞版本的实现。

|  |
| --- |
| **// 代码清单4-1 未使用Netty 的阻塞网络编程**  **public class** PlainOioServer {  **public void** serve(**int** port) **throws** IOException {  *// 将服务器绑定到指定端口* **final** ServerSocket socket = **new** ServerSocket(port);  **try** {  **for** (; ; ) {  *// 接受连接* **final** Socket clientSocket = socket.accept();  System.***out***.println(**"Accepted connection from "** + clientSocket);  *// 创建一个新的线程来处理该连接* **new** Thread(**new** Runnable() {  @Override  **public void** run() {  OutputStream out;  **try** {  *// 将消息写给已连接的客户端* out = clientSocket.getOutputStream();  out.write(**"Hi!\r\n"**.getBytes(Charset.*forName*(**"UTF-8"**)));  out.flush();  clientSocket.close();  } **catch** (IOException e) {  e.printStackTrace();  } **finally** {  **try** {  *// 关闭连接* clientSocket.close();  } **catch** (IOException ex) {  *// ignore on close* }  }  }  }).start(); *// 启动线程* }  } **catch** (IOException e) {  e.printStackTrace();  }  } } |

这段代码完全可以处理中等数量的并发客户端。但是随着应用程序变得流行起来，你会发现它并不能很好地伸缩到支撑成千上万的并发连入连接。你决定改用异步网络编程，但是很快就发现异步API 是完全不同的，以至于现在你不得不重写你的应用程序。其非阻塞版本如代码清单4-2 所示。