# 【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。

|  |
| --- |
| **// 代码清单4-2 未使用Netty 的异步网络编程**  **public class** PlainNioServer {  **public void** serve(**int** port) **throws** IOException {  ServerSocketChannel serverChannel = ServerSocketChannel.*open*();  serverChannel.configureBlocking(**false**);  ServerSocket ss = serverChannel.socket();  InetSocketAddress address = **new** InetSocketAddress(port);  *// 将服务器绑定到选定的端口* ss.bind(address);  *// 打开Selector来处理Channel* Selector selector = Selector.*open*();  *// 将ServerSocket注册到Selector 以接受连接* serverChannel.register(selector, SelectionKey.***OP\_ACCEPT***);  **final** ByteBuffer msg = ByteBuffer.*wrap*(**"Hi!\r\n"**.getBytes());  **for** (; ; ) {  **try** {  selector.select();  } **catch** (IOException ex) {  ex.printStackTrace();  *//handle exception* **break**;  }  *// 获取所有接收事件的SelectionKey 实例* Set<SelectionKey> readyKeys = selector.selectedKeys();  Iterator<SelectionKey> iterator = readyKeys.iterator();  **while** (iterator.hasNext()) {  SelectionKey key = iterator.next();  iterator.remove();  **try** {  *// 检查事件是否是一个新的已经就绪可以被接受的连接* **if** (key.isAcceptable()) {  ServerSocketChannel server = (ServerSocketChannel) key.channel();  SocketChannel client = server.accept();  client.configureBlocking(**false**);  *// 接受客户端，并将它注册到选择器* client.register(selector, SelectionKey.***OP\_WRITE*** | SelectionKey.***OP\_READ***, msg.duplicate());  System.***out***.println(**"Accepted connection from "** + client);  }   *// 检查套接字是否已经准备好写数据* **if** (key.isWritable()) {  SocketChannel client = (SocketChannel) key.channel();  ByteBuffer buffer = (ByteBuffer) key.attachment();  **while** (buffer.hasRemaining()) {  *// 将数据写到已连接的客户端* **if** (client.write(buffer) == **0**) {  **break**;  }  }   *// 关闭连接* client.close();  }  } **catch** (IOException ex) {  key.cancel();  **try** {  key.channel().close();  } **catch** (IOException cex) {  *// ignore on close* }  }  }  }  } } |

如同你所看到的，虽然这段代码所做的事情与之前的版本完全相同，但是代码却截然不同。如果为了用于非阻塞I/O 而重新实现这个简单的应用程序，都需要一次完全的重写的话，那么不难想象，移植真正复杂的应用程序需要付出什么样的努力。

鉴于此，让我们来看看使用Netty 实现该应用程序将会是什么样子吧。

### 4.1.2 通过Netty 使用OIO 和NIO

我们将先编写这个应用程序的另一个阻塞版本，这次我们将使用Netty 框架，如代码清单4-3所示。

|  |
| --- |
| **// 代码清单4-3 使用Netty 的阻塞网络处理**  **public class** NettyOioServer {  **public void** server(**int** port) **throws** Exception {  **final** ByteBuf buf = Unpooled.*unreleasableBuffer*(Unpooled.*copiedBuffer*(**"Hi!\r\n"**, Charset.*forName*(**"UTF-8"**)));  EventLoopGroup group = **new** OioEventLoopGroup();  **try** {  *// 创建ServerBootstrap* ServerBootstrap b = **new** ServerBootstrap();  b.group(group)  *// 使用OioEventLoopGroup以允许阻塞模式（旧的I/O）* .channel(OioServerSocketChannel.**class**)  .localAddress(**new** InetSocketAddress(port))  *// 指定ChannelInitializer，对于每个已接受的连接都调用它* .childHandler(**new** ChannelInitializer<SocketChannel>() {  @Override  **public void** initChannel(SocketChannel ch) **throws** Exception {  *// 添加一个ChannelInboundHandlerAdapter 以拦截和处理事件* ch.pipeline().addLast(**new** ChannelInboundHandlerAdapter() {  @Override  **public void** channelActive(ChannelHandlerContext ctx) **throws** Exception {  *// 将消息写到客户端，并添加ChannelFutureListener，以便消息一被写完就关闭连接* ctx.writeAndFlush(buf.duplicate()).addListener(ChannelFutureListener.***CLOSE***);  }  });  }  });  *// 绑定服务器接受连接* ChannelFuture f = b.bind().sync();  f.channel().closeFuture().sync();  } **finally** {  *// 释放所有的资源* group.shutdownGracefully().sync();  }  } } |

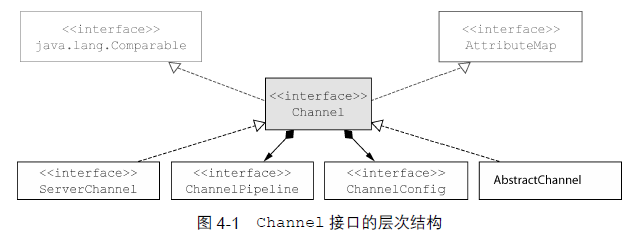
接下来，我们使用Netty 和非阻塞I/O 来实现同样的逻辑。

|  |
| --- |
| **// 代码清单4-4 使用Netty 的异步网络处理**  **public class** NettyNioServer {  **public void** server(**int** port) **throws** Exception {  **final** ByteBuf buf = Unpooled.*unreleasableBuffer*(Unpooled.*copiedBuffer*(**"Hi!\r\n"**, Charset.*forName*(**"UTF-8"**)));  *// 为非阻塞模式使用NioEventLoopGroup* NioEventLoopGroup group = **new** NioEventLoopGroup();  **try** {  *// 创建ServerBootstrap* ServerBootstrap b = **new** ServerBootstrap();  b.group(group)  *// 指定ChannelInitializer，对于每个已接受的连接都调用它* .channel(NioServerSocketChannel.**class**)  .localAddress(**new** InetSocketAddress(port))  *// 添加一个ChannelInboundHandlerAdapter 以拦截和处理事件* .childHandler(**new** ChannelInitializer<SocketChannel>() {  @Override  **public void** initChannel(SocketChannel ch) **throws** Exception {  ch.pipeline().addLast(**new** ChannelInboundHandlerAdapter() {  @Override  **public void** channelActive(ChannelHandlerContext ctx) **throws** Exception {  *// 将消息写到客户端，并添加ChannelFutureListener，以便消息一被写完就关闭连接* ctx.writeAndFlush(buf.duplicate()).addListener(ChannelFutureListener.***CLOSE***);  }  });  }  });  *// 绑定服务器以接受连接* ChannelFuture f = b.bind().sync();  f.channel().closeFuture().sync();  } **finally** {  *// 释放所有的资源* group.shutdownGracefully().sync();  }  } } |

因为Netty 为每种传输的实现都暴露了相同的API，所以无论选用哪一种传输的实现，你的代码都仍然几乎不受影响。在所有的情况下，传输的实现都依赖于interface Channel、ChannelPipeline 和ChannelHandler。

## 4.2 传输API

传输API 的核心是interface Channel，它被用于所有的I/O 操作。Channel 类的层次结构如图4-1 所示。



如图所示，每个Channel 都将会被分配一个ChannelPipeline 和ChannelConfig。ChannelConfig 包含了该Channel 的所有配置设置，并且支持热更新。由于特定的传输可能具有独特的设置，所以它可能会实现一个ChannelConfig 的子类型。（请参考ChannelConfig实现对应的Javadoc。）

由于Channel 是独一无二的，所以为了保证顺序将Channel 声明为java.lang.Comparable 的一个子接口。因此，如果两个不同的Channel 实例都返回了相同的散列码，那么AbstractChannel 中的compareTo()方法的实现将会抛出一个Error。

ChannelPipeline 持有所有将应用于入站和出站数据以及事件的ChannelHandler 实例，这些ChannelHandler 实现了应用程序用于处理状态变化以及数据处理的逻辑。

ChannelHandler 的典型用途包括：

* 将数据从一种格式转换为另一种格式；
* 提供异常的通知；
* 提供Channel 变为活动的或者非活动的通知；
* 提供当Channel 注册到EventLoop 或者从EventLoop 注销时的通知；
* 提供有关用户自定义事件的通知。

拦截过滤器 ChannelPipeline 实现了一种常见的设计模式—拦截过滤器（InterceptingFilter）。UNIX 管道是另外一个熟悉的例子：多个命令被链接在一起，其中一个命令的输出端将连接到命令行中下一个命令的输入端。

你也可以根据需要通过添加或者移除ChannelHandler实例来修改ChannelPipeline。通过利用Netty的这项能力可以构建出高度灵活的应用程序。例如，每当STARTTLS协议被请求时， 你可以简单地通过向ChannelPipeline 添加一个适当的ChannelHandler（SslHandler）来按需地支持STARTTLS协议。

除了访问所分配的ChannelPipeline 和ChannelConfig 之外，也可以利用Channel的其他方法，其中最重要的列举在表4-1 中。



稍后我们将进一步深入地讨论所有这些特性的应用。目前，请记住，Netty 所提供的广泛功能只依赖于少量的接口。这意味着，你可以对你的应用程序逻辑进行重大的修改，而又无需大规模地重构你的代码库。

考虑一下写数据并将其冲刷到远程节点这样的常规任务。代码清单4-5 演示了使用Channel.writeAndFlush()来实现这一目的。