

第 2 章 你的第一款 Netty 应用程序



本章主要内容

- 设置开发环境
- 编写 Echo 服务器和客户端
- 构建并测试应用程序

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。这能让你对在第 3 章中开始的对 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。

我们将假设，你想要捣鼓示例代码，并且想很快就开始编写自己的代码。虽然你可以使用纯文本编辑器，但是我们仍然强烈地建议你使用用于 Java 的集成开发环境（IDE）。

2.1.1 获取并安装 Java 开发工具包

你的操作系统可能已经安装了 JDK。为了找到答案，可以在命令行输入：

```
javac -version
```

如果得到的是javac 1.7……或者1.8……，则说明已经设置好了并且可以略过此步^①。

否则，请从 <http://java.com/en/download/manual.jsp> 处获取 JDK 第 8 版。请留心，需要下载的是 JDK，而不是 Java 运行时环境 (JRE)，其只可以运行 Java 应用程序，但是不能够编译它们。该网站为每个平台都提供了可执行的安装程序。如果需要安装说明，可以在同一个网站上找到相关的信息。

建议执行以下操作：

- 将环境变量 JAVA_HOME 设置为你的 JDK 安装位置（在 Windows 上，默认值将类似于 C:\Program Files\Java\jdk1.8.0_121）；
- 将%JAVA_HOME%\bin（在 Linux 上为\${JAVA_HOME}/bin）添加到你的执行路径。

2.1.2 下载并安装 IDE

下面是使用最广泛的 Java IDE，都可以免费获取：

- Eclipse—— www.eclipse.org；
- NetBeans—— www.netbeans.org；
- IntelliJ IDEA Community Edition—— www.jetbrains.com。

所有这 3 种对我们使用的构建工具 Apache Maven 都拥有完整的支持。NetBeans 和 IntelliJ IDEA 都通过可执行的安装程序进行分发。Eclipse 通常使用 Zip 归档文件进行分发，当然也有一些自定义的版本包含了自安装程序。

2.1.3 下载和安装 Apache Maven

即使你已经熟悉 Maven 了，我们仍然建议你至少大致浏览一下这一节。

Maven 是一款广泛使用的由 Apache 软件基金会 (ASF) 开发的构建管理工具。Netty 项目以及本书的示例都使用了它。构建和运行这些示例并不需要你成为一个 Maven 专家，但是如果想要对其进行扩展，我们推荐你阅读附录中的 Maven 简介。

你需要安装 Maven 吗

Eclipse 和 NetBeans^②自带了一个内置的 Maven 安装包，对于我们的目的来说开箱即可工作得良好。如果你将要在一个拥有它自己的 Maven 存储库的环境中工作，那么你的配置管理员可能就有一个预先配置好的能配合它使用的 Maven 安装包。

在本书中文版出版时，Maven 的最新版本是 3.3.9。你可以从 <http://maven.apache.org/download.cgi> 下载适用于你的操作系统的 tar.gz 或者 zip 归档文件^③。安装很简单：将归档文件的所

① Netty 的一组受限特性可以运行于 JDK 1.6，但是 JDK 8 或者更高版本则是编译时必需的，包括运行最新版本的 Maven。

② 包括 IntelliJ IDEA。——译者注

③ 也可以通过 HomeBrew 或者 Scoop 来安装 Maven，更加简单方便。——译者注

有内容解压到你所选择的任意的文件夹（我们将其称为<安装目录>）。这将创建目录<安装目录>\apache-maven-3.3.9。

和设置 Java 环境一样：

- 将环境变量 M2_HOME 设置为指向<安装目录>\apache-maven-3.3.9；
- 将%M2_HOME%\bin（或者在 Linux 上为\${M2_HOME}/bin）添加到你的执行路径。

这将使得你可以通过在命令行执行 mvn.bat（或者 mvn）来运行 Maven。

2.1.4 配置工具集

如果你已经按照推荐设置了环境变量 JAVA_HOME 和 M2_HOME，那么你可能会发现，当你启动自己的 IDE 时，它已经发现了你的 Java 和 Maven 的安装位置。如果你需要进行手动配置，我们所列举的所有 IDE 版本在 Preferences 或者 Settings 下都有设置这些变量的菜单项。相关的细节请查阅文档。

这就完成了开发环境的配置。在接下来的各节中，我们将介绍你要构建的第一个 Netty 应用程序的详细信息，同时我们将更加深入地了解该框架的 API。之后，你就能使用刚刚设置好的工具来构建和运行 Echo 服务器和客户端了。

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。

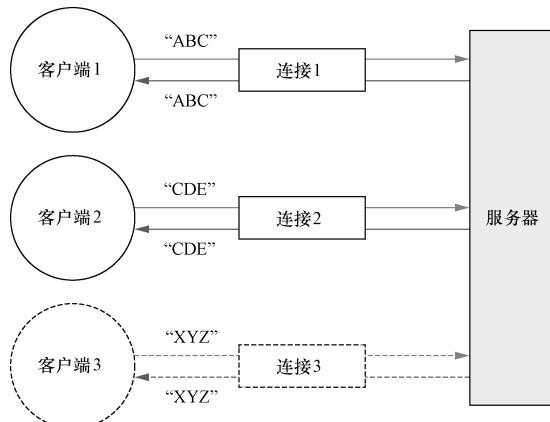


图 2-1 Echo 客户端和服务器

虽然我们已经谈到了客户端，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。

我们将从考察服务器端代码开始这个项目。

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 ChannelInboundHandlerAdapter 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 channelRead() 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

```
@Sharable
public class EchoServerHandler extends ChannelInboundHandlerAdapter {
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) {
        ByteBuf in = (ByteBuf) msg;
        System.out.println(
            "Server received: " + in.toString(CharsetUtil.UTF_8));
    }
}
```

将消息记录到控制台

标示一个 ChannelHandler 可以被多个 Channel 安全地共享

```

        ctx.write(in);
    }

    @Override
    public void channelReadComplete(ChannelHandlerContext ctx) {
        ctx.writeAndFlush(Unpooled.EMPTY_BUFFER)
            .addListener(ChannelFutureListener.CLOSE);
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx,
        Throwable cause) {
        cause.printStackTrace();
        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 ChannelHandler。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。(6.4 节详细地讨论了异常处理)。

除了 ChannelInboundHandlerAdapter 之外，还有很多需要学习的 ChannelHandler 的子类型和实现，我们将在第 6 章和第 7 章中对它们进行详细的阐述。目前，请记住下面这些关键点：

- 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。

我们将在第 4 章对传输进行详细的讨论。

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.length != 1) {
            System.err.println(
                "Usage: " + EchoServer.class.getSimpleName() +
                " <port>");
        }
        int port = Integer.parseInt(args[0]);
        new EchoServer(port).start();
    }
}

创建 Event-LoopGroup ①
public void start() throws Exception {
    final EchoServerHandler serverHandler = new EchoServerHandler();
    EventLoopGroup group = new NioEventLoopGroup();
    try {
        ServerBootstrap b = new ServerBootstrap();
        b.group(group)
            .channel(NioServerSocketChannel.class)
            .localAddress(new InetSocketAddress(port))
            .childHandler(new ChannelInitializer<SocketChannel>(){
                @Override
                public void initChannel(SocketChannel ch)
                    throws Exception {
                        ch.pipeline().addLast(serverHandler);
                }
            });
    }
}

创建 Server-Bootstrap ②
使用指定的端口设置套接字地址 ④
添加一个 EchoServer-Handler 到子 Channel 的 ChannelPipeline ⑤
指定所使用的 NIO 传输 Channel ③
设置端口值（如果端口参数的格式不正确，则抛出一个NumberFormatException）
```

① 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。——译者注

```

    ChannelFuture f = b.bind().sync();
    f.channel().closeFuture().sync();
} finally {
    group.shutdownGracefully().sync(); ⑥
}
} ⑦
} ⑧
}

    ↑
    获取 Channel 的
    CloseFuture，并
    且阻塞当前线
    程直到它完成

    ↑
    关闭 EventLoopGroup,
    释放所有的资源

```

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 NioEventLoopGroup①来接受和处理新的连接，并且将 Channel 的类型指定为 NioServerSocketChannel③。在此之后，你将本地地址设置为一个具有选定端口的 InetSocketAddress④。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，而且我们将在第 3 章中对相关的抽象（EventLoopGroup、SocketChannel 和 ChannelInitializer）进行详细的讨论。

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⑦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⑧。

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。我们将在第 4 章中对传输进行更加详细的探讨。

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。

在这个时候，服务器已经初始化，并且已经就绪能被使用了。在下一节中，我们将探讨对应的客户端应用程序的代码。

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()`^①——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`

```
@Sharable
public class EchoClientHandler extends
    SimpleChannelInboundHandler<ByteBuf> {
    @Override
    public void channelActive(ChannelHandlerContext ctx) {
        ctx.writeAndFlush(Unpooled.copiedBuffer("Netty rocks!", CharsetUtil.UTF_8));
    }
    @Override
    public void channelRead0(ChannelHandlerContext ctx, ByteBuf in) {
        System.out.println(
            "Client received: " + in.toString(CharsetUtil.UTF_8));
    }
    @Override
```

The code listing is annotated with three callout boxes:

- A box on the right side of the first code block contains the text: "标记该类的实例可以被多个 Channel 共享". An arrow points from the word "Sharable" to this text.
- A box on the right side of the second code block contains the text: "当被通知 Channel 是活跃的时候，发送一条消息". An arrow points from the word "channelActive" to this text.
- A box on the right side of the third code block contains the text: "记录已接收消息的转储". An arrow points from the word "channelRead0" to this text.

^① `SimpleChannelInboundHandler` 的 `channelRead0()` 方法的相关讨论参见 <https://github.com/netty/netty/wiki/New-and-noteworthy-in-5.0#channelread0--messagereceived>，其中 Netty5 的开发工作已经关闭。——译者注

```

public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) {
    cause.printStackTrace();
    ctx.close();
}
}

```

在发生异常时，
记录错误并关闭
Channel

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 `EchoServerHandler` 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)。

第 5 章和第 6 章将对消息的资源管理进行详细的介绍。

2.4.2 引导客户端

如同将在代码清单 2-4 中所看到的，引导客户端类似于引导服务器，不同的是，客户端是使用主机和端口参数来连接远程地址，也就是这里的 Echo 服务器的地址，而不是绑定到一个一直被监听的端口。

代码清单 2-4 客户端的主类

```

public class EchoClient {
    private final String host;
    private final int port;

    public EchoClient(String host, int port) {
        this.host = host;
        this.port = port;
    }

    public void start() throws Exception {
        EventLoopGroup group = new NioEventLoopGroup();
        try {
            Bootstrap b = new Bootstrap();
            b.group(group)
                .channel(NioSocketChannel.class)
                .remoteAddress(new InetSocketAddress(host, port))
                .handler(new ChannelInitializer<SocketChannel>() {
                    @Override
                    public void initChannel(SocketChannel ch)
                        throws Exception {
                        ch.pipeline().addLast(
                            new EchoClientHandler());
                    }
                });
            ChannelFuture f = b.connect().sync();
            f.channel().closeFuture().sync();
        } finally {
            group.shutdownGracefully().sync();
        }
    }

    public static void main(String[] args) throws Exception {
        if (args.length != 2) {
            System.err.println(
                "Usage: " + EchoClient.class.getSimpleName() +
                " <host> <port>");
            return;
        }

        String host = args[0];
        int port = Integer.parseInt(args[1]);
        new EchoClient(host, port).start();
    }
}

```

设置服务器的 InetSocketAddress

在创建 Channel 时，向 ChannelPipeline 中添加一个 EchoClientHandler 实例

创建 Bootstrap

指定 EventLoopGroup 以处理客户端事件；需要适用于 NIO 的实现

适用于 NIO 传输的 Channel 类型

连接到远程节点，阻塞等待直到连接完成

阻塞，直到 Channel 关闭

关闭线程池并且释放所有的资源

和之前一样，使用了 NIO 传输。注意，你可以在客户端和服务器上分别使用不同的传输。例如，在服务器端使用 NIO 传输，而在客户端使用 OIO 传输。在第 4 章，我们将探讨影响你选择适用于特定用例的特定传输的各种因素和场景。

让我们回顾一下这一节中所介绍的要点：

- 为初始化客户端，创建了一个 Bootstrap 实例；

- 为进行事件处理分配了一个 `NioEventLoopGroup` 实例，其中事件处理包括创建新的连接以及处理入站和出站数据；
 - 为服务器连接创建了一个 `InetSocketAddress` 实例；
 - 当连接被建立时，一个 `EchoClientHandler` 实例会被安装到（该 `Channel` 的）`ChannelPipeline` 中；
 - 在一切都设置完成后，调用 `Bootstrap.connect()` 方法连接到远程节点；
- 完成了客户端，你便可以着手构建并测试该系统了。

2.5 构建和运行 Echo 服务器和客户端

在这一节中，我们将介绍编译和运行 Echo 服务器和客户端所需的所有步骤。

Echo 客户端/服务器的 Maven 工程

这本书的附录使用 Echo 客户端/服务器工程的配置，详细地解释了多模块 Maven 工程是如何组织的。这部分内容对于构建和运行该应用程序来说并不是必读的，之所以推荐阅读这部分内容，是因为它能帮助你更好地理解本书的示例以及 Netty 项目本身。

2.5.1 运行构建

要构建 Echo 客户端和服务器，请进入到代码示例根目录下的 chapter2 目录执行以下命令：

```
mvn clean package
```

这将产生非常类似于代码清单 2-5 所示的输出（我们已经编辑忽略了几个构建过程中的非必要步骤）。

代码清单 2-5 构建 Echo 客户端和服务器

```
[INFO] Scanning for projects...
[INFO] -----
[INFO] Reactor Build Order:
[INFO]
[INFO] Chapter 2. Your First Netty Application - Echo App
[INFO] Chapter 2. Echo Client
[INFO] Chapter 2. Echo Server
[INFO]
[INFO] -----
[INFO] Building Chapter 2. Your First Netty Application - 2.0-SNAPSHOT
[INFO] -----
[INFO]
[INFO] --- maven-clean-plugin:2.6.1:clean (default-clean) @ chapter2 ---
[INFO]
[INFO] -----
[INFO] Building Chapter 2. Echo Client 2.0-SNAPSHOT
```

```
[INFO] -----
[INFO]
[INFO] --- maven-clean-plugin:2.6.1:clean (default-clean)
  @ echo-client ---
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources)
  @ echo-client ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 1 resource
[INFO]
[INFO] --- maven-compiler-plugin:3.3:compile (default-compile)
  @ echo-client ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 2 source files to
  \netty-in-action\chapter2\Client\target\classes
[INFO]
[INFO] --- maven-resources-plugin:2.6:testResources (default-testResources)
  @ echo-client ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory
  \netty-in-action\chapter2\Client\src\test\resources
[INFO]
[INFO] --- maven-compiler-plugin:3.3:testCompile (default-testCompile)
  @ echo-client ---
[INFO] No sources to compile
[INFO]
[INFO] --- maven-surefire-plugin:2.18.1:test (default-test)
  @ echo-client ---
[INFO] No tests to run.
[INFO]
[INFO] --- maven-jar-plugin:2.6:jar (default-jar) @ echo-client ---
[INFO] Building jar:
  \netty-in-action\chapter2\Client\target\echo-client-2.0-SNAPSHOT.jar
[INFO]
[INFO] -----
[INFO] Building Chapter 2. Echo Server 2.0-SNAPSHOT
[INFO] -----
[INFO]
[INFO] --- maven-clean-plugin:2.6.1:clean (default-clean)
  @ echo-server ---
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources)
  @ echo-server ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 1 resource
[INFO]
[INFO] --- maven-compiler-plugin:3.3:compile (default-compile)
  @ echo-server ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 2 source files to
  \netty-in-action\chapter2\Server\target\classes
[INFO]
[INFO] --- maven-resources-plugin:2.6:testResources (default-testResources)
  @ echo-server ---
```

```
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory
  \netty-in-action\chapter2\Server\src\test\resources
[INFO]
[INFO] --- maven-compiler-plugin:3.3:testCompile (default-testCompile)
  @ echo-server ---
[INFO] No sources to compile
[INFO]
[INFO] --- maven-surefire-plugin:2.18.1:test (default-test)
  @ echo-server ---
[INFO] No tests to run.
[INFO]
[INFO] --- maven-jar-plugin:2.6:jar (default-jar) @ echo-server ---
[INFO] Building jar:
  \netty-in-action\chapter2\Server\target\echo-server-2.0-SNAPSHOT.jar
[INFO] -----
[INFO] Reactor Summary:
[INFO]
[INFO] Chapter 2. Your First Netty Application ... SUCCESS [ 0.134 s]
[INFO] Chapter 2. Echo Client ..... SUCCESS [ 1.509 s]
[INFO] Chapter 2. Echo Ser..... SUCCESS [ 0.139 s]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 1.886 s
[INFO] Finished at: 2015-11-18T17:14:10-05:00
[INFO] Final Memory: 18M/216M
[INFO] -----
```

下面是前面的构建日志中记录的主要步骤：

- Maven 确定了构建顺序：首先是父 pom.xml，然后是各个模块（子工程）；
- 如果在用户的本地存储库中没有找到 Netty 构件，Maven 将从公共的 Maven 存储库中下载它们（此处未显示）；
- 运行了构建生命周期中的 clean 和 compile 阶段；
- 最后执行了 maven-jar-plugin。

Maven Reactor 的摘要显示所有的项目都已经被成功地构建。两个子工程的目标目录的文件列表现在应该类似于代码清单 2-6。

代码清单 2-6 构建的构件列表

```
Directory of nia\chapter2\Client\target
03/16/2015 09:45 PM <DIR>      classes
03/16/2015 09:45 PM           5,614 echo-client-1.0-SNAPSHOT.jar
03/16/2015 09:45 PM <DIR>      generated-sources
03/16/2015 09:45 PM <DIR>      maven-archiver
03/16/2015 09:45 PM <DIR>      maven-status

Directory of nia\chapter2\Server\target
03/16/2015 09:45 PM <DIR>      classes
03/16/2015 09:45 PM           5,629 echo-server-1.0-SNAPSHOT.jar
```

```
03/16/2015 09:45 PM <DIR> generated-sources  
03/16/2015 09:45 PM <DIR> maven-archiver  
03/16/2015 09:45 PM <DIR> maven-status
```

2.5.2 运行 Echo 服务器和客户端

要运行这些应用程序组件，可以直接使用 Java 命令。但是在 POM 文件中，已经为你配置好了 exec-maven-plugin 来做这个（参见附录以获取详细信息）。

并排打开两个控制台窗口，一个进到 chapter2\Server 目录中，另外一个进到 chapter2\Client 目录中。

在服务器的控制台中执行这个命令：

```
mvn exec:java
```

应该会看到类似于下面的内容：

```
[INFO] Scanning for projects...  
[INFO]  
[INFO] -----  
[INFO] Building Echo Server 1.0-SNAPSHOT  
[INFO] -----  
[INFO]  
[INFO] >>> exec-maven-plugin:1.2.1:java (default-cli) >  
      validate @ echo-server >>>  
[INFO]  
[INFO] <<< exec-maven-plugin:1.2.1:java (default-cli) <  
      validate @ echo-server <<<  
[INFO]  
[INFO] --- exec-maven-plugin:1.2.1:java (default-cli) @ echo-server ---  
      nia.chapter2.echoserver.EchoServer  
      started and listening for connections on /0:0:0:0:0:0:0:9999
```

服务器现在已经启动并准备好接受连接。现在在客户端的控制台中执行同样的命令：

```
mvn exec:java
```

应该会看到下面的内容：

```
[INFO] Scanning for projects...  
[INFO]  
[INFO] -----  
[INFO] Building Echo Client 1.0-SNAPSHOT  
[INFO] -----  
[INFO]  
[INFO] >>> exec-maven-plugin:1.2.1:java (default-cli) >  
      validate @ echo-client >>>  
[INFO]  
[INFO] <<< exec-maven-plugin:1.2.1:java (default-cli) <  
      validate @ echo-client <<<  
[INFO]  
[INFO] --- exec-maven-plugin:1.2.1:java (default-cli) @ echo-client ---  
      Client received: Netty rocks!
```

```
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 2.833 s
[INFO] Finished at: 2015-03-16T22:03:54-04:00
[INFO] Final Memory: 10M/309M
[INFO] -----
```

同时在服务器的控制台中，应该会看到这个：

```
Server received: Netty rocks!
```

每次运行客户端时，在服务器的控制台中你都能看到这条日志语句。

下面是发生的事：

- (1) 一旦客户端建立连接，它就发送它的消息——Netty rocks!；
- (2) 服务器报告接收到的消息，并将其回送给客户端；
- (3) 客户端报告返回的消息并退出。

你所看到的都是预期的行为，现在让我们看看故障是如何被处理的。服务器应该还在运行，所以在服务器的控制台中按下 Ctrl+C 来停止该进程。一旦它停止，就再次使用下面的命令启动客户端：

```
mvn exec:java
```

代码清单 2-7 展示了你应该会从客户端的控制台中看到的当它不能连接到服务器时的输出。

代码清单 2-7 Echo 客户端的异常处理

```
[INFO] Scanning for projects...
[INFO]
[INFO] -----
[INFO] Building Echo Client 1.0-SNAPSHOT
[INFO] -----
[INFO]
[INFO] >>> exec-maven-plugin:1.2.1:java (default-cli) >
      validate @ echo-client >>>
[INFO]
[INFO] <<< exec-maven-plugin:1.2.1:java (default-cli) <
      validate @ echo-client <<<
[INFO]
[INFO] --- exec-maven-plugin:1.2.1:java (default-cli) @ echo-client ---
[WARNING]
java.lang.reflect.InvocationTargetException
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    ...
Caused by: java.net.ConnectException: Connection refused:
no further information: localhost/127.0.0.1:9999
    at sun.nio.ch.SocketChannelImpl.checkConnect(Native Method)
    at sun.nio.ch.SocketChannelImpl
        .finishConnect(SocketChannelImpl.java:739)
    at io.netty.channel.socket.nio.NioSocketChannel
        .doFinishConnect(NioSocketChannel.java:208)
```

```
at io.netty.channel.nio
    .AbstractNioChannel$AbstractNioUnsafe
        .finishConnect(AbstractNioChannel.java:281)
at io.netty.channel.nio.NioEventLoop
    .processSelectedKey(NioEventLoop.java:528)
at io.netty.channel.nio.NioEventLoop.
    processSelectedKeysOptimized(NioEventLoop.java:468)
at io.netty.channel.nio.NioEventLoop
    .processSelectedKeys(NioEventLoop.java:382)
at io.netty.channel.nio.NioEventLoop
    .run(NioEventLoop.java:354)
at io.netty.util.concurrent.SingleThreadEventExecutor$2
    .run(SingleThreadEventExecutor.java:116)
at io.netty.util.concurrent.DefaultThreadFactory
    $DefaultRunnableDecorator.run(DefaultThreadFactory.java:137)
    .
[INFO] -----
[INFO] BUILD FAILURE
[INFO] -----
[INFO] Total time: 3.801 s
[INFO] Finished at: 2015-03-16T22:11:16-04:00
[INFO] Final Memory: 10M/309M
[INFO] -----
[ERROR] Failed to execute goal org.codehaus.mojo:
    exec-maven-plugin:1.2.1:java (default-cli) on project echo-client:
        An exception occurred while executing the Java class. null:
        InvocationTargetException: Connection refused:
            no further information: localhost/127.0.0.1:9999 -> [Help 1]
```

发生了什么？客户端试图连接服务器，其预期运行在 `localhost:9999` 上。但是连接失败了（和预期的一样），因为服务器在这之前就已经停止了，所以在客户端导致了一个 `java.net.ConnectException`。这个异常触发了 `EchoClientHandler` 的 `exceptionCaught()` 方法，打印出了栈跟踪并关闭了 `Channel`（见代码清单 2-3）。

2.6 小结

在本章中，你设置好了开发环境，并且构建和运行了你的第一款 Netty 客户端和服务器。虽然这只是一个简单的应用程序，但是它可以伸缩到支持数千个并发连接——每秒可以比普通的基于套接字的 Java 应用程序处理多得多的消息。

在接下来的几章中，你将看到更多关于 Netty 如何简化可伸缩性和并发性的例子。我们也将更加深入地了解 Netty 对于关注点分离的架构原则的支持。通过提供正确的抽象来解耦业务逻辑和网络编程逻辑，Netty 使得可以很容易地跟上快速演化的需求，而又不危及系统的稳定性。

在下一章中，我们将提供对 Netty 体系架构的概述。这将为你在后续的章节中对 Netty 的内部进行深入而全面的学习提供上下文。

第 3 章 Netty 的组件和设计

本章主要内容

- Netty 的技术和体系结构方面的内容
- Channel、EventLoop 和 ChannelFuture
- ChannelHandler 和 ChannelPipeline
- 引导

在第 1 章中，我们给出了 Java 高性能网络编程的历史以及技术基础的小结。这为 Netty 的核心概念和构件块的概述提供了背景。

在第 2 章中，我们把我们的讨论范围扩大到了应用程序的开发。通过构建一个简单的客户端和服务器，你学习了引导，并且获得了最重要的 ChannelHandler API 的实战经验。与此同时，你也验证了自己的开发工具都能正常运行。

由于本书剩下的部分都建立在这份材料的基础之上，所以我们将从两个不同的但却又密切相关的视角来探讨 Netty：类库的视角以及框架的视角。对于使用 Netty 编写高效的、可重用的和可维护的代码来说，两者缺一不可。

从高层次的角度来看，Netty 解决了两个相应的关注领域，我们可将其大致标记为技术的和体系结构的。首先，它的基于 Java NIO 的异步的和事件驱动的实现，保证了高负载下应用程序性能的最大化和可伸缩性。其次，Netty 也包含了一组设计模式，将应用程序逻辑从网络层解耦，简化了开发过程，同时也最大限度地提高了可测试性、模块化以及代码的可重用性。

在我们更加详细地研究 Netty 的各个组件时，我们将密切关注它们是如何通过协作来支撑这些体系结构上的最佳实践的。通过遵循同样的原则，我们便可获得 Netty 所提供的所有益处。牢记这个目标，在本章中，我们将回顾到目前为止我们介绍过的概念和组件。

3.1 Channel、EventLoop 和 ChannelFuture

接下来的各节将会为我们对于 Channel、EventLoop 和 ChannelFuture 类进行的讨论增添更多的细节，这些类合在一起，可以被认为是 Netty 网络抽象的代表：

- Channel——Socket；
- EventLoop——控制流、多线程处理、并发；
- ChannelFuture——异步通知。

3.1.1 Channel 接口

基本的 I/O 操作 (`bind()`、`connect()`、`read()` 和 `write()`) 依赖于底层网络传输所提供的原语。在基于 Java 的网络编程中，其基本的构造是 class `Socket`。Netty 的 Channel 接口所提供的 API，大大地降低了直接使用 `Socket` 类的复杂性。此外，Channel 也是拥有许多预定义的、专门化实现的广泛类层次结构的根，下面是一个简短的部分清单：

- `EmbeddedChannel`；
- `LocalServerChannel`；
- `NioDatagramChannel`；
- `NioSctpChannel`；
- `NioSocketChannel`。

3.1.2 EventLoop 接口

`EventLoop` 定义了 Netty 的核心抽象，用于处理连接的生命周期中所发生的事件。我们将在第 7 章中结合 Netty 的线程处理模型的上下文对 `EventLoop` 进行详细的讨论。目前，图 3-1 在高层次上说明了 `Channel`、`EventLoop`、`Thread` 以及 `EventLoopGroup` 之间的关系。

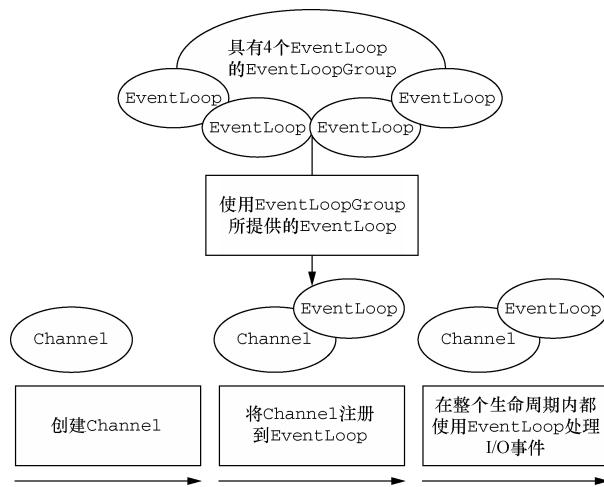


图 3-1 Channel、EventLoop 和 EventLoopGroup

这些关系是：

- 一个 EventLoopGroup 包含一个或者多个 EventLoop；
- 一个 EventLoop 在它的生命周期内只和一个 Thread 绑定；
- 所有由 EventLoop 处理的 I/O 事件都将在它专有的 Thread 上被处理；
- 一个 Channel 在它的生命周期内只注册于一个 EventLoop；
- 一个 EventLoop 可能会被分配给一个或多个 Channel。

注意，在这种设计中，一个给定 Channel 的 I/O 操作都是由相同的 Thread 执行的，实际上消除了对于同步的需要。

3.1.3 ChannelFuture 接口

正如我们已经解释过的那样，Netty 中所有的 I/O 操作都是异步的。因为一个操作可能不会立即返回，所以我们需要一种用于在之后的某个时间点确定其结果的方法。为此，Netty 提供了 ChannelFuture 接口，其 addListener() 方法注册了一个 ChannelFutureListener，以便在某个操作完成时（无论是否成功）得到通知。

关于 ChannelFuture 的更多讨论 可以将 ChannelFuture 看作是将来要执行的操作的结果的占位符。它究竟什么时候被执行则可能取决于若干的因素，因此不可能准确地预测，但是可以肯定的是它将会被执行。此外，所有属于同一个 Channel 的操作都被保证其将以它们被调用的顺序被执行。

我们将在第 7 章中深入地讨论 EventLoop 和 EventLoopGroup。

3.2 ChannelHandler 和 ChannelPipeline

现在，我们将更加细致地看一看那些管理数据流以及执行应用程序处理逻辑的组件。

3.2.1 ChannelHandler 接口

从应用程序开发人员的角度来看，Netty 的主要组件是 ChannelHandler，它充当了所有处理入站和出站数据的应用程序逻辑的容器。这是可行的，因为 ChannelHandler 的方法是由网络事件（其中术语“事件”的使用非常广泛）触发的。事实上，ChannelHandler 可专门用于几乎任何类型的动作，例如将数据从一种格式转换为另外一种格式，或者处理转换过程中所抛出的异常。

举例来说，ChannelInboundHandler 是一个你将会经常实现的子接口。这种类型的 ChannelHandler 接收入站事件和数据，这些数据随后将会被你的应用程序的业务逻辑所处理。当你要给连接的客户端发送响应时，也可以从 ChannelInboundHandler 冲刷数据。你的应用程序的业务逻辑通常驻留在一个或者多个 ChannelInboundHandler 中。

3.2.2 ChannelPipeline 接口

ChannelPipeline 提供了 ChannelHandler 链的容器，并定义了用于在该链上传播入站和出站事件流的 API。当 Channel 被创建时，它会被自动地分配到它专属的 ChannelPipeline。

ChannelHandler 安装到 ChannelPipeline 中的过程如下所示：

- 一个 ChannelInitializer 的实现被注册到了 ServerBootstrap 中^①；
- 当 ChannelInitializer.initChannel() 方法被调用时，ChannelInitializer 将在 ChannelPipeline 中安装一组自定义的 ChannelHandler；
- ChannelInitializer 将它自己从 ChannelPipeline 中移除。

为了审查发送或者接收数据时将会发生什么，让我们来更加深入地研究 ChannelPipeline 和 ChannelHandler 之间的共生关系吧。

ChannelHandler 是专为支持广泛的用途而设计的，可以将它看作是处理往来 ChannelPipeline 事件（包括数据）的任何代码的通用容器。图 3-2 说明了这一点，其展示了从 ChannelHandler 派生的 ChannelInboundHandler 和 ChannelOutboundHandler 接口。

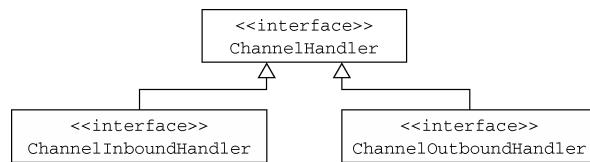


图 3-2 ChannelHandler 类的层次结构

使得事件流经 ChannelPipeline 是 ChannelHandler 的工作，它们是在应用程序的初始化或者引导阶段被安装的。这些对象接收事件、执行它们所实现的处理逻辑，并将数据传递给链中的下一个 ChannelHandler。它们的执行顺序是由它们被添加的顺序所决定的。实际上，被我们称为 ChannelPipeline 的是这些 ChannelHandler 的编排顺序。

图 3-3 说明了一个 Netty 应用程序中入站和出站数据流之间的区别。从一个客户端应用程序的角度来看，如果事件的运动方向是从客户端到服务器端，那么我们称这些事件为出站的，反之则称为入站的。

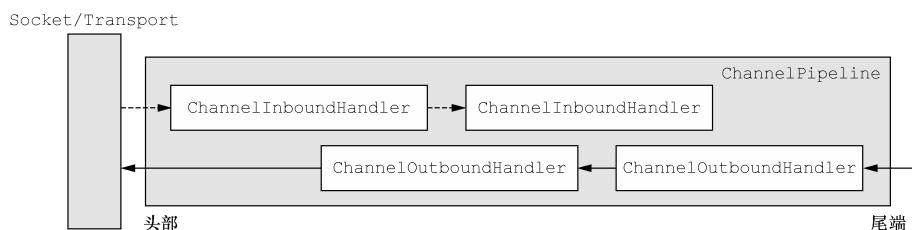


图 3-3 包含入站和出站 ChannelHandler 的 ChannelPipeline

^① 或者用于客户端的 Bootstrap。——译者注

图 3-3 也显示了入站和出站 ChannelHandler 可以被安装到同一个 ChannelPipeline 中。如果一个消息或者任何其他的人站事件被读取，那么它会从 ChannelPipeline 的头部开始流动，并被传递给第一个 ChannelInboundHandler。这个 ChannelHandler 不一定会实际地修改数据，具体取决于它的具体功能，在这之后，数据将会被传递给链中的下一个 ChannelInboundHandler。最终，数据将会到达 ChannelPipeline 的尾端，届时，所有处理就都结束了。

数据的出站运动（即正在被写的数据）在概念上也是一样的。在这种情况下，数据将从 ChannelOutboundHandler 链的尾端开始流动，直到它到达链的头部为止。在这之后，出站数据将会到达网络传输层，这里显示为 Socket。通常情况下，这将触发一个写操作。

关于入站和出站 ChannelHandler 的更多讨论

通过使用作为参数传递到每个方法的 ChannelHandlerContext，事件可以被传递给当前 ChannelHandler 链中的下一个 ChannelHandler。因为你有时会忽略那些不感兴趣的事件，所以 Netty 提供了抽象基类 ChannelInboundHandlerAdapter 和 ChannelOutboundHandlerAdapter。通过调用 ChannelHandlerContext 上的对应方法，每个都提供了简单地将事件传递给下一个 ChannelHandler 的方法的实现。随后，你可以通过重写你所感兴趣的那些方法来扩展这些类。

鉴于出站操作和入站操作是不同的，你可能会想知道如果将两个类别的 ChannelHandler 都混合添加到同一个 ChannelPipeline 中会发生什么。虽然 ChannelInboundHandle 和 ChannelOutboundHandle 都扩展自 ChannelHandler，但是 Netty 能区分 ChannelInboundHandler 实现和 ChannelOutboundHandler 实现，并确保数据只会在具有相同定向类型的两个 ChannelHandler 之间传递。

当 ChannelHandler 被添加到 ChannelPipeline 时，它将会被分配一个 ChannelHandlerContext，其代表了 ChannelHandler 和 ChannelPipeline 之间的绑定。虽然这个对象可以被用于获取底层的 Channel，但是它主要还是被用于写出站数据。

在 Netty 中，有两种发送消息的方式。你可以直接写到 Channel 中，也可以写到和 ChannelHandler 相关联的 ChannelHandlerContext 对象中。前一种方式将会导致消息从 ChannelPipeline 的尾端开始流动，而后者将导致消息从 ChannelPipeline 中的下一个 ChannelHandler 开始流动。

3.2.3 更加深入地了解 ChannelHandler

正如我们之前所说的，有许多不同类型的 ChannelHandler，它们各自的功能主要取决于它们的超类。Netty 以适配器类的形式提供了大量默认的 ChannelHandler 实现，其旨在简化应用程序处理逻辑的开发过程。你已经看到了，ChannelPipeline 中的每个 ChannelHandler 将负责把事件转发到链中的下一个 ChannelHandler。这些适配器类（及它们的子类）将自动执行这个操作，所以你可以只重写那些你想要特殊处理的方法和事件。

为什么需要适配器类

有一些适配器类可以将编写自定义的 ChannelHandler 所需要的努力降到最低限度，因为它们提供了定义在对应接口中的所有方法的默认实现。

下面这些是编写自定义 ChannelHandler 时经常会用到的适配器类：

- ChannelHandlerAdapter
- ChannelInboundHandlerAdapter
- ChannelOutboundHandlerAdapter
- ChannelDuplexHandler

接下来我们将研究 3 个 ChannelHandler 的子类型：编码器、解码器和 SimpleChannelInboundHandler<T>——ChannelInboundHandlerAdapter 的一个子类。

3.2.4 编码器和解码器

当你通过 Netty 发送或者接收一个消息的时候，就将会发生一次数据转换。入站消息会被解码；也就是说，从字节转换为另一种格式，通常是一个 Java 对象。如果是出站消息，则会发生相反方向的转换：它将从它的当前格式被编码为字节。这两种方向的转换的原因很简单：网络数据总是一系列的字节。

对应于特定的需要，Netty 为编码器和解码器提供了不同类型的抽象类。例如，你的应用程序可能使用了一种中间格式，而不需要立即将消息转换成字节。你将仍然需要一个编码器，但是它将派生自一个不同的超类。为了确定合适的编码器类型，你可以应用一个简单的命名约定。

通常来说，这些基类的名称将类似于 ByteToMessageDecoder 或 MessageToByteEncoder。对于特殊的类型，你可能会发现类似于 ProtobufEncoder 和 ProtobufDecoder 这样的名称——预置的用来支持 Google 的 Protocol Buffers。

严格地说，其他的处理器也可以完成编码器和解码器的功能。但是，正如有用来简化 ChannelHandler 的创建的适配器类一样，所有由 Netty 提供的编码器/解码器适配器类都实现了 ChannelOutboundHandler 或者 ChannelInboundHandler 接口。

你将会发现对于入站数据来说，channelRead 方法/事件已经被重写了。对于每个从入站 Channel 读取的消息，这个方法都将会被调用。随后，它将调用由预置解码器所提供的 decode() 方法，并将已解码的字节转发给 ChannelPipeline 中的下一个 ChannelInboundHandler。

出站消息的模式是相反方向的：编码器将消息转换为字节，并将它们转发给下一个 ChannelOutboundHandler。

3.2.5 抽象类 SimpleChannelInboundHandler

最常见的情况是，你的应用程序会利用一个 ChannelHandler 来接收解码消息，并对该数

据应用业务逻辑。要创建一个这样的 ChannelHandler，你只需要扩展基类 SimpleChannelInboundHandler<T>，其中 T 是你要处理的消息的 Java 类型。在这个 ChannelHandler 中，你将需要重写基类的一个或者多个方法，并且获取一个到 ChannelHandlerContext 的引用，这个引用将作为输入参数传递给 ChannelHandler 的所有方法。

在这种类型的 ChannelHandler 中，最重要的方法是 channelRead0(ChannelHandlerContext, T)。**除了要求不要阻塞当前的 I/O 线程之外**，其具体实现完全取决于你。我们稍后将对这一主题进行更多的说明。

3.3 引导

Netty 的引导类为应用程序的网络层配置提供了容器，这涉及将一个进程绑定到某个指定的端口，或者将一个进程连接到另一个运行在某个指定主机的指定端口上的进程。

通常来说，我们把前面的用例称作引导一个服务器，后面的用例称作引导一个客户端。虽然这个术语简单方便，但是它略微掩盖了一个重要的事实，即“服务器”和“客户端”实际上表示了不同的网络行为；换句话说，是监听传入的连接还是建立到一个或者多个进程的连接。

面向连接的协议 请记住，严格来说，“连接”这个术语仅适用于面向连接的协议，如 TCP，其保证了两个连接端点之间消息的有序传递。

因此，有两种类型的引导：一种用于客户端（简单地称为 `Bootstrap`），而另一种 (`ServerBootstrap`) 用于服务器。无论你的应用程序使用哪种协议或者处理哪种类型的数据，唯一决定它使用哪种引导类的是它是作为一个客户端还是作为一个服务器。表 3-1 比较了这两种类型的引导类。

表 3-1 比较 `Bootstrap` 类

类 别	<code>Bootstrap</code>	<code>ServerBootstrap</code>
网络编程中的作用	连接到远程主机和端口	绑定到一个本地端口
<code>EventLoopGroup</code> 的数目	1	2 ^①

这两种类型的引导类之间的第一个区别已经讨论过了：`ServerBootstrap` 将绑定到一个端口，因为服务器必须要监听连接，而 `Bootstrap` 则是由想要连接到远程节点的客户端应用程序所使用的。

第二个区别可能更加明显。引导一个客户端只需要一个 `EventLoopGroup`，但是一个 `ServerBootstrap` 则需要两个（也可以是同一个实例）。为什么呢？

^① 实际上，`ServerBootstrap` 类也可以只使用一个 `EventLoopGroup`，此时其将在两个场景下共用同一个 `EventLoopGroup`。——译者注

因为服务器需要两组不同的 Channel。第一组将只包含一个 ServerChannel，代表服务器自身的已绑定到某个本地端口的正在监听的套接字。而第二组将包含所有已创建的用来处理传入客户端连接（对于每个服务器已经接受的连接都有一个）的 Channel。图 3-4 说明了这个模型，并且展示了为何需要两个不同的 EventLoopGroup。

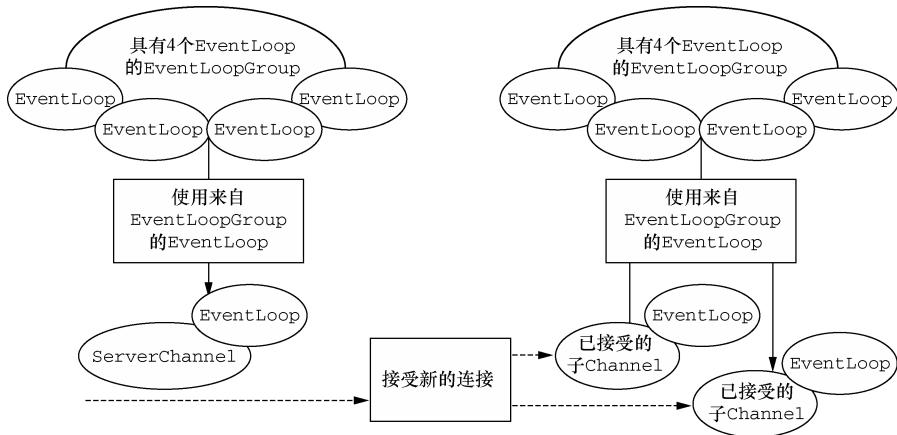


图 3-4 具有两个 EventLoopGroup 的服务器

与 ServerChannel 相关联的 EventLoopGroup 将分配一个负责为传入连接请求创建 Channel 的 EventLoop。一旦连接被接受，第二个 EventLoopGroup 就会给它的 Channel 分配一个 EventLoop。

3.4 小结

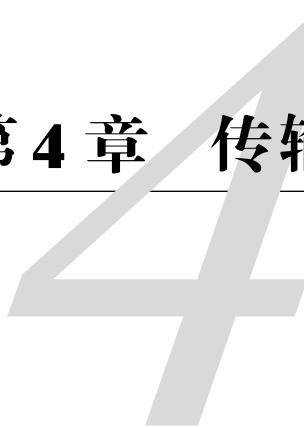
在本章中，我们从技术和体系结构这两个角度探讨了理解 Netty 的重要性。我们也更加详细地重新审视了之前引入的一些概念和组件，特别是 ChannelHandler、ChannelPipeline 和引导。

特别地，我们讨论了 ChannelHandler 类的层次结构，并介绍了编码器和解码器，描述了它们在数据和网络字节格式之间来回转换的互补功能。

下面的许多章节都将致力于深入研究这些组件，而这里所呈现的概览应该有助于你对整体的把控。

下一章将探索 Netty 所提供的不同类型的传输，以及如何选择一个最适合于你的应用程序的传输。

第 4 章 传输



本章主要内容

- OIO——阻塞传输
- NIO——异步传输
- Local——JVM 内部的异步通信
- Embedded——测试你的 ChannelHandler

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。

如果你有 Java 网络编程的经验，那么你可能已经发现，在某些时候，你需要支撑比预期多很多的并发连接。如果你随后尝试从阻塞传输切换到非阻塞传输，那么你可能会因为这两种网络 API 的截然不同而遇到问题。

然而，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。简而言之，你可以将时间花在其他更有成效的事情上。

在本章中，我们将学习这个通用 API，并通过和 JDK 的对比来证明它极其简单易用。我们将阐述 Netty 自带的不同传输实现，以及它们各自适用的场景。有了这些信息，你会发现选择最适合于你的应用程序的选项将是直截了当的。

本章的唯一前提是 Java 编程语言的相关知识。有网络框架或者网络编程相关的经验更好，但不是必需的。

我们先来看一看传输在现实世界中是如何工作的。

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。如果你曾享受过使用 JDK 进行网络编程的乐趣，那么这段代码将唤起你美好的回忆。

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 {
        final ServerSocket socket = new ServerSocket(port);           | 将服务器绑定
        try {                                                       | 到指定端口
            for (;;) {
                final Socket clientSocket = socket.accept();         | 接受连接
                System.out.println(
                    "Accepted connection from " + clientSocket);
                new Thread(new Runnable() {
                    @Override
                    public void run() {
                        OutputStream out;
                        try {
                            out = clientSocket.getOutputStream();
                            out.write("Hi!\r\n".getBytes(
                                Charset.forName("UTF-8")));
                            out.flush();
                            clientSocket.close();
                        }
                        catch (IOException e) {
                            e.printStackTrace();
                        }
                    finally {
                        try {
                            clientSocket.close();                         | 关闭连接
                        }
                        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 ssocket = serverChannel.socket();
        InetSocketAddress address = new InetSocketAddress(port);
        ssocket.bind(address);
        Selector selector = Selector.open();
        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;
            }
            Set<SelectionKey> readyKeys = selector.selectedKeys();
            Iterator<SelectionKey> iterator = readyKeys.iterator();
            while (iterator.hasNext()) {
                SelectionKey key = iterator.next();
                iterator.remove();
                try {
                    if (key.isAcceptable()) {
                        ServerSocketChannel server =
                            (ServerSocketChannel)key.channel();
                        SocketChannel client = server.accept();
                        client.configureBlocking(false);
                        client.register(selector, SelectionKey.OP_WRITE |
                            SelectionKey.OP_READ, msg.duplicate());
                        System.out.println(
                            "Accepted connection from " + client);
                    }
                    if (key.isWritable()) {
                        SocketChannel client =
                            (SocketChannel)key.channel();
                        ByteBuffer buffer =
                            (ByteBuffer)key.attachment();
                        while (buffer.hasRemaining()) {
                            if (client.write(buffer) == 0) {
                                break;
                            }
                        }
                        client.close();
                    }
                } catch (IOException ex) {
                    key.cancel();
                    try {
                        key.channel().close();
                    }
                }
            }
        }
    }
}

```

打开 Selector 来处理 Channel

将 ServerSocket 注册到 Selector 以接受连接

获取所有接收事件的 Selection-Key 实例

接受客户端，并将它注册到选择器

将服务器绑定到选定的端口

等待需要处理的新事件；阻塞将一直持续到下一个传入事件

检查事件是否是一个新的已经就绪可以被接受的连接

检查套接字是否已经准备好写数据

将数据写到已连接的客户端

关闭连接

如同你所看到的，虽然这段代码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 {
            ServerBootstrap b = new ServerBootstrap();
            b.group(group)
              .channel(OioServerSocketChannel.class)
              .localAddress(new InetSocketAddress(port))
              .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 {
                                  ctx.writeAndFlush(buf.duplicate())
                                      .addListener(
                                          ChannelFutureListener.CLOSE);
                              }
                          });
                  }
              });
            ChannelFuture f = b.bind().sync();
            f.channel().closeFuture().sync();
        } finally {
            group.shutdownGracefully();
        }
    }
}

```

创建 Server-Bootstrap

指定 Channel-Initializer, 对于每个已接受的连接都调用它

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

使用 OioEventLoopGroup 以允许阻塞模式(旧的 I/O)

添加一个 Channel-InboundHandler-Adapter 以拦截和处理事件

绑定服务器以接受连接

```

        } finally {
            group.shutdownGracefully().sync();   <-- 释放所有的资源
        }
    }
}

```

接下来，我们使用 Netty 和非阻塞 I/O 来实现同样的逻辑。

4.1.3 非阻塞的 Netty 版本

代码清单 4-4 和代码清单 4-3 几乎一模一样，除了高亮显示的那两行。这就是从阻塞 (OIO) 传输切换到非阻塞 (NIO) 传输需要做的所有变更。

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.copiedBuffer("Hi!\r\n",
            Charset.forName("UTF-8"));
        EventLoopGroup group = new NioEventLoopGroup();           <-- 为非阻塞模式使用
                                                                NioEventLoopGroup
        try {
指定 Channel-          |   ServerBootstrap b = new ServerBootstrap();           <-- 创建
Initializer，对于      |   b.group(group).channel(NioServerSocketChannel.class)  | ServerBootstrap
每个已接受的          |       .localAddress(new InetSocketAddress(port))         |
连接都调用它          |       .childHandler(new ChannelInitializer<SocketChannel>() {  |
                                                                |           @Override
添加 ChannelInbound-  |               public void initChannel(SocketChannel ch)     |
HandlerAdapter 以接收 |                   throws Exception{                     |
和处理事件          |                   ch.pipeline().addLast(                  |
                                                                |                       new ChannelInboundHandlerAdapter() {  |
                                                                |                           @Override
                                                                |                           public void channelActive(  |
                                                                |                               ChannelHandlerContext ctx) throws Exception {  |
                                                                |                               ctx.writeAndFlush(buf.duplicate())  |
                                                                |                               .addListener(                    |
                                                                |                                   ChannelFutureListener.CLOSE);  |
                                                                |                           }
                                                                |                   });
                                                                |               });
                                                                |           });
                                                                |       );
                                                                |       ChannelFuture f = b.bind().sync();
                                                                |       f.channel().closeFuture().sync();
                                                                |   } finally {
                                                                |       group.shutdownGracefully().sync();   <-- 释放所有的
                                                                |   }
                                                                | }
}

```

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。

在看过一些使用基于 Netty 的传输的这些优点之后，让我们仔细看看传输 API 本身。

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。

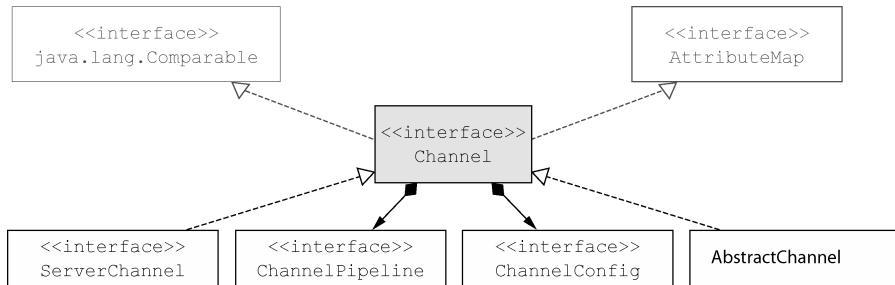


图 4-1 Channel 接口的层次结构

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 (Intercepting Filter)。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^①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。

表4-1 Channel的方法

方 法 名	描 述
eventLoop	返回分配给Channel的EventLoop
pipeline	返回分配给Channel的ChannelPipeline
isActive	如果Channel是活动的，则返回true。活动的意义可能依赖于底层的传输。例如，一个Socket传输一旦连接到了远程节点便是活动的，而一个Datagram传输一旦被打开便是活动的
localAddress	返回本地的SocketAddress
remoteAddress	返回远程的SocketAddress
write	将数据写到远程节点。这个数据将被传递给ChannelPipeline，并且排队直到它被冲刷
flush	将之前已写的数据冲刷到底层传输，如一个Socket
writeAndFlush	一个简便的方法，等同于调用write()并接着调用flush()

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。

代码清单4-5 写出到Channel

```
Channel channel = ...
→ByteBuf buf = Unpooled.copiedBuffer("your data", CharsetUtil.UTF_8);
    ChannelFuture cf = channel.writeAndFlush(buf); ←写数据并
    cf.addListener(new ChannelFutureListener() {           冲刷它
        @Override
        public void operationComplete(ChannelFuture future) { ←添加 ChannelFutureListener 以便在写操作完成后接收通知
            if (future.isSuccess()) { ←创建持有要写的数据的 ByteBuf
                System.out.println("Write successful"); ←写操作完成，并且没有错误发生
            } else {
        }
    }
}
```

① 参见 STARTTLS: <http://en.wikipedia.org/wiki/STARTTLS>。

```

        System.err.println("Write error");
        future.cause().printStackTrace();
    }
}
);

```

Netty 的 Channel 实现是线程安全的，因此你可以存储一个到 Channel 的引用，并且每当你需要向远程节点写数据时，都可以使用它，即使当时许多线程都在使用它。代码清单 4-6 展示了一个多线程写数据的简单例子。需要注意的是，消息将被保证按顺序发送。

代码清单 4-6 从多个线程使用同一个 Channel

```

final Channel channel = ...
final ByteBuf buf = Unpooled.copiedBuffer("your data",
    CharsetUtil.UTF_8).retain();           ← 创建持有要写数据的
Runnable writer = new Runnable() {          ByteBuf
    @Override                           ← 创建将数据写到
    public void run() {                  Channel 的 Runnable
        channel.writeAndFlush(buf.duplicate());
    }
};

Executor executor = Executors.newCachedThreadPool();           ← 获取到线程池
// write in one thread                      Executor 的引用
executor.execute(writer);                         ← 递交写任务给线程池以便
// write in another thread                   在某个线程中执行
executor.execute(writer);                         ← 递交另一个写任务以便
...                                              在另一个线程中执行

```

4.3 内置的传输

Netty 内置了一些可开箱即用的传输。因为并不是它们所有的传输都支持每一种协议，所以你必须选择一个和你的应用程序所使用的协议相容的传输。在本节中我们将讨论这些关系。

表 4-2 显示了所有 Netty 提供的传输。

表 4-2 Netty 所提供的传输

名 称	包	描 述
NIO	io.netty.channel.socket.nio	使用 java.nio.channels 包作为基础——基于选择器的方式
Epoll ^①	io.netty.channel.epoll	由 JNI 驱动的 epoll() 和非阻塞 IO。这个传输支持只有在 Linux 上可用的多种特性，如 SO_REUSEPORT，比 NIO 传输更快，而且是完全非阻塞的

① 这个是 Netty 特有的实现，更加适配 Netty 现有的线程模型，具有更高的性能以及更低的垃圾回收压力，详见 <https://github.com/netty/netty/wiki/Native-transports>。——译者注

续表

名 称	包	描 述
OIO	io.netty.channel.socket.nio	使用 java.net 包作为基础——使用阻塞流
Local	io.netty.channel.local	可以在 VM 内部通过管道进行通信的本地传输
Embedded	io.netty.channel.embedded	Embedded 传输，允许使用 ChannelHandler 而又不需要一个真正的基于网络的传输。这在测试你的 ChannelHandler 实现时非常有用

我们将在接下来的几节中详细讨论这些传输。

4.3.1 NIO——非阻塞 I/O

NIO 提供了一个所有 I/O 操作的全异步的实现。它利用了自 NIO 子系统被引入 JDK 1.4 时便可用的基于选择器的 API。

选择器背后的基本概念是充当一个注册表，在那里你将可以请求在 Channel 的状态发生变化时得到通知。可能的状态变化有：

- 新的 Channel 已被接受并且就绪；
- Channel 连接已经完成；
- Channel 有已经就绪的可供读取的数据；
- Channel 可用于写数据。

选择器运行在一个检查状态变化并对其做出相应响应的线程上，在应用程序对状态的改变做出响应之后，选择器将会被重置，并将重复这个过程。

表 4-3 中的常量值代表了由 class java.nio.channels.SelectionKey 定义的位模式。这些位模式可以组合起来定义一组应用程序正在请求通知的状态变化集。

表 4-3 选择操作的位模式

名 称	描 述
OP_ACCEPT	请求在接受新连接并创建 Channel 时获得通知
OP_CONNECT	请求在建立一个连接时获得通知
OP_READ	请求当数据已经就绪，可以从 Channel 中读取时获得通知
OP_WRITE	请求当可以向 Channel 中写更多的数据时获得通知。这处理了套接字缓冲区被完全填满时的情况，这种情况通常发生在数据的发送速度比远程节点可处理的速度更快的时候

对于所有 Netty 的传输实现都共有的用户级别 API 完全地隐藏了这些 NIO 的内部细节。图 4-2 展示了该处理流程。

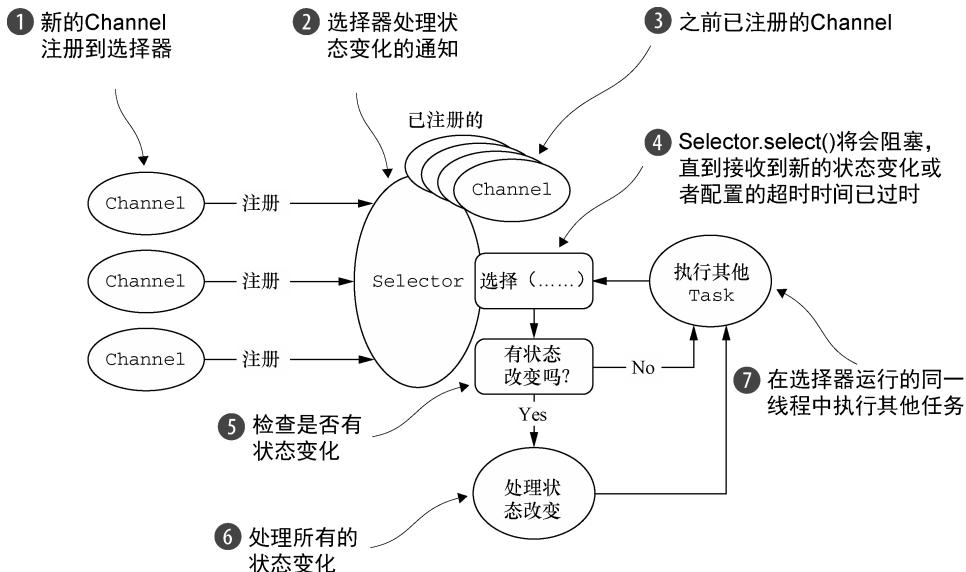


图 4-2 选择并处理状态的变化

零拷贝

零拷贝（zero-copy）是一种目前只有在使用 NIO 和 Epoll 传输时才可使用的特性。它使你可以快速高效地将数据从文件系统移动到网络接口，而不需要将其从内核空间复制到用户空间，其在像 FTP 或者 HTTP 这样的协议中可以显著地提升性能。但是，并不是所有的操作系统都支持这一特性。特别地，它对于实现了数据加密或者压缩的文件系统是不可用的——只能传输文件的原始内容。反过来说，传输已被加密的文件则不是问题。

4.3.2 Epoll——用于 Linux 的本地非阻塞传输

正如我们之前所说的，Netty 的 NIO 传输基于 Java 提供的异步/非阻塞网络编程的通用抽象。虽然这保证了 Netty 的非阻塞 API 可以在任何平台上使用，但它也包含了相应的限制，因为 JDK 为了在所有系统上提供相同的功能，必须做出妥协。

Linux 作为高性能网络编程的平台，其重要性与日俱增，这催生了大量先进特性的开发，其中包括 epoll——一个高度可扩展的 I/O 事件通知特性。这个 API 自 Linux 内核版本 2.5.44 (2002) 被引入，提供了比旧的 POSIX `select` 和 `poll` 系统调用^①更好的性能，同时现在也是 Linux 上非阻塞网络编程的事实标准。Linux JDK NIO API 使用了这些 epoll 调用。

① 参见 Linux 手册页中的 `epoll(4)`: <http://linux.die.net/man/4/epoll>。

Netty为Linux提供了一组NIO API，其以一种和它本身的设计更加一致的方式使用epoll，并且以一种更加轻量的方式使用中断。^①如果你的应用程序旨在运行于Linux系统，那么请考虑利用这个版本的传输；你将发现在高负载下它的性能要优于JDK的NIO实现。

这个传输的语义与在图 4-2 所示的完全相同，而且它的用法也是简单直接的。相关示例参照代码清单 4-4。如果要在那个代码清单中使用 epoll 替代 NIO，只需要将 `NioEventLoopGroup` 替换为 `EpollEventLoopGroup`，并且将 `NioServerSocketChannel.class` 替换为 `EpollServerSocketChannel.class` 即可。

4.3.3 OIO——旧的阻塞 I/O

Netty 的 OIO 传输实现代表了一种折中：它可以通过常规的传输 API 使用，但是由于它是建立在 `java.net` 包的阻塞实现之上的，所以它不是异步的。但是，它仍然非常适合于某些用途。

例如，你可能需要移植使用了一些进行阻塞调用的库（如 JDBC^②）的遗留代码，而将逻辑转换为非阻塞的可能也是不切实际的。相反，你可以在短期内使用 Netty 的 OIO 传输，然后再将你的代码移植到纯粹的异步传输上。让我们来看一看怎么做。

在 `java.net` API 中，你通常会有一个用来接受到达正在监听的 `ServerSocket` 的新连接的线程。会创建一个新的和远程节点进行交互的套接字，并且会分配一个新的用于处理相应通信流量的线程。这是必需的，因为某个指定套接字上的任何 I/O 操作在任意的时间点上都可能会阻塞。使用单个线程来处理多个套接字，很容易导致一个套接字上的阻塞操作也捆绑了所有其他的套接字。

有了这个背景，你可能会想，Netty 是如何能够使用和用于异步传输相同的 API 来支持 OIO 的呢。答案就是，Netty 利用了 `SO_TIMEOUT` 这个 `Socket` 标志，它指定了等待一个 I/O 操作完成的最大毫秒数。如果操作在指定的时间间隔内没有完成，则将会抛出一个 `SocketTimeoutException`。Netty 将捕获这个异常并继续处理循环。在 `EventLoop` 下一次运行时，它将再次尝试。这实际上也是类似于 Netty 这样的异步框架能够支持 OIO 的唯一方式^③。图 4-3 说明了这个逻辑。

4.3.4 用于 JVM 内部通信的 Local 传输

Netty 提供了一个 Local 传输，用于在同一个 JVM 中运行的客户端和服务器程序之间的异步通信。同样，这个传输也支持对于所有 Netty 传输实现都共同的 API。

① JDK 的实现是水平触发，而 Netty 的（默认的）是边沿触发。有关的详细信息参见 epoll 在维基百科上的解释：http://en.wikipedia.org/wiki/Epoll-Triggering_modes。

② JDBC 的文档可以在 www.oracle.com/technetwork/java/javase/jdbc/index.html 获取。

③ 这种方式的一个问题是，当一个 `SocketTimeoutException` 被抛出时填充栈跟踪所需的时间，其对于性能来说代价很大。

在这个传输中，和服务器 Channel 相关联的 `SocketAddress` 并没有绑定物理网络地址；相反，只要服务器还在运行，它就会被存储在注册表里，并在 Channel 关闭时注销。因为这个传输并不接受真正的网络流量，所以它并不能够和其他传输实现进行互操作。因此，客户端希望连接到（在同一个 JVM 中）使用了这个传输的服务器端时也必须使用它。除了这个限制，它的使用方式和其他的传输一模一样。

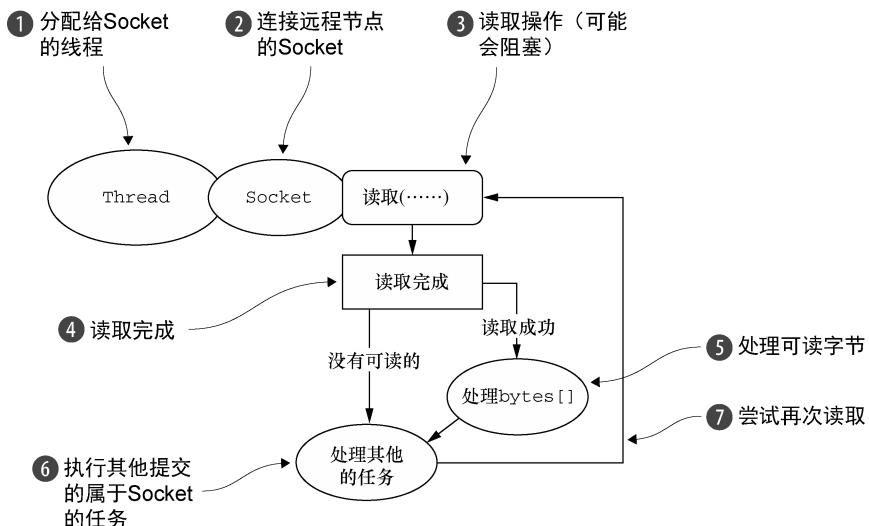


图 4-3 OIO 的处理逻辑

4.3.5 Embedded 传输

Netty 提供了一种额外的传输，使得你可以将一组 `ChannelHandler` 作为帮助器类嵌入到其他的 `ChannelHandler` 内部。通过这种方式，你将可以扩展一个 `ChannelHandler` 的功能，而又不需要修改其内部代码。

不足为奇的是，`Embedded` 传输的关键是一个被称为 `EmbeddedChannel` 的具体的 `Channel` 实现。在第 9 章中，我们将详细地讨论如何使用这个类来为 `ChannelHandler` 的实现创建单元测试用例。

4.4 传输的用例

既然我们已经详细了解了所有的传输，那么让我们考虑一下选用一个适用于特定用途的协议的因素吧。正如前面所提到的，并不是所有的传输都支持所有的核心协议，其可能会限制你的选择。表 4-4 展示了截止出版时的传输和其所支持的协议。

表 4-4 支持的传输和网络协议

传 输	TCP	UDP	SCTP*	UDT ^①
NIO	×	×	×	×
Epoll (仅 Linux)	×	×	—	—
OIO	×	×	×	×

* 参见 RFC 2960 中有关流控制传输协议 (SCTP) 的解释：www.ietf.org/rfc/rfc2960.txt。

在 Linux 上启用 SCTP

SCTP 需要内核的支持，并且需要安装用户库。

例如，对于 Ubuntu，可以使用下面的命令：

```
# sudo apt-get install libssctp1
```

对于 Fedora，可以使用 yum：

```
#sudo yum install kernel-modules-extra.x86_64 lksctp-tools.x86_64
```

有关如何启用 SCTP 的详细信息，请参考你的 Linux 发行版的文档。

虽然只有 SCTP 传输有这些特殊要求，但是其他传输可能也有它们自己的配置选项需要考虑。此外，如果只是为了支持更高的并发连接数，服务器平台可能需要配置得和客户端不一样。

这里是一些你很可能遇到的用例。

- 非阻塞代码库——如果你的代码库中没有阻塞调用（或者你能够限制它们的范围），那么在 Linux 上使用 NIO 或者 epoll 始终是个好主意。虽然 NIO epoll 旨在处理大量的并发连接，但是在处理较小数目的并发连接时，它也能很好地工作，尤其是考虑到它在连接之间共享线程的方式。
- 阻塞代码库——正如我们已经指出的，如果你的代码库严重地依赖于阻塞 I/O，而且你的应用程序也有一个相应的设计，那么在你尝试将其直接转换为 Netty 的 NIO 传输时，你将可能会遇到和阻塞操作相关的问题。不要为此而重写你的代码，可以考虑分阶段迁移：先从 OIO 开始，等你的代码修改好之后，再迁移到 NIO（或者使用 epoll，如果你在使用 Linux）。
- 在同一个 JVM 内部的通信——在同一个 JVM 内部的通信，不需要通过网络暴露服务，是 Local 传输的完美用例。这将消除所有真实网络操作的开销，同时仍然使用你的 Netty 代码库。如果随后需要通过网络暴露服务，那么你将只需要把传输改为 NIO 或者 OIO 即可。
- 测试你的 ChannelHandler 实现——如果你想要为自己的 ChannelHandler 实现编写单元测试，那么请考虑使用 Embedded 传输。这既便于测试你的代码，而又不需要创建大量的模拟（mock）对象。你的类将仍然符合常规的 API 事件流，保证该 ChannelHandler 在和真实的传输一起使用时能够正确地工作。你将在第 9 章中发现关于测试

① UDT 协议实现了基于 UDP 协议的可靠传输，详见 <https://zh.wikipedia.org/zh-cn/UDT>。——译者注

ChannelHandler 的更多信息。

表 4-5 总结了我们探讨过的用例。

表 4-5 应用程序的最佳传输

应用程序的需求	推荐的传输
非阻塞代码库或者一个常规的起点	NIO (或者在 Linux 上使用 epoll)
阻塞代码库	OIO
在同一个 JVM 内部的通信	Local
测试 ChannelHandler 的实现	Embedded

4.5 小结

在本章中，我们研究了传输、它们的实现和使用，以及 Netty 是如何将它们呈现给开发者的。

我们深入探讨了 Netty 预置的传输，并且解释了它们的行为。因为不是所有的传输都可以在相同的 Java 版本下工作，并且其中一些可能只在特定的操作系统下可用，所以我们也描述了它们的最低需求。最后，我们讨论了你可以如何匹配不同的传输和特定用例的需求。

在下一章中，我们将关注于 ByteBuf 和 ByteBufHolder——Netty 的数据容器。我们将展示如何使用它们以及如何通过它们获得最佳性能。

第 5 章 ByteBuf

本章主要内容

- ByteBuf——Netty 的数据容器
- API 的详细信息
- 用例
- 内存分配

正如前面所提到的，网络数据的基本单位总是字节。Java NIO 提供了 ByteBuffer 作为它的字节容器，但是这个类使用起来过于复杂，而且也有些繁琐。

Netty 的 ByteBuffer 替代品是 ByteBuf，一个强大的实现，既解决了 JDK API 的局限性，又为网络应用程序的开发者提供了更好的 API。

在本章中我们将会说明和 JDK 的 ByteBuffer 相比，ByteBuf 的卓越功能性和灵活性。这也将有助于更好地理解 Netty 数据处理的一般方式，并为将在第 6 章中针对 ChannelPipeline 和 ChannelHandler 的讨论做好准备。

5.1 ByteBuf 的 API

Netty 的数据处理 API 通过两个组件暴露——abstract class ByteBuf 和 interface ByteBufHolder。

下面是一些 ByteBuf API 的优点：

- 它可以被用户自定义的缓冲区类型扩展；
- 通过内置的复合缓冲区类型实现了透明的零拷贝；
- 容量可以按需增长（类似于 JDK 的 `StringBuilder`）；
- 在读和写这两种模式之间切换不需要调用 `ByteBuffer` 的 `flip()` 方法；
- 读和写使用了不同的索引；
- 支持方法的链式调用；

- 支持引用计数；
- 支持池化。

其他类可用于管理 ByteBuf 实例的分配，以及执行各种针对于数据容器本身和它所持有的数据的操作。我们将在仔细研究 ByteBuf 和 ByteBufHolder 时探讨这些特性。

5.2 ByteBuf 类——Netty 的数据容器

因为所有的网络通信都涉及字节序列的移动，所以高效易用的数据结构明显是必不可少的。Netty 的 ByteBuf 实现满足并超越了这些需求。让我们首先来看看它是如何通过使用不同的索引来简化对它所包含的数据的访问的吧。

5.2.1 它是如何工作的

ByteBuf 维护了两个不同的索引：一个用于读取，一个用于写入。当你从 ByteBuf 读取时，它的 readerIndex 将会被递增已经被读取的字节数。同样地，当你写入 ByteBuf 时，它的 writerIndex 也会被递增。图 5-1 展示了一个空 ByteBuf 的布局结构和状态。



图 5-1 一个读索引和写索引都设置为 0 的 16 字节 ByteBuf

要了解这些索引两两之间的关系，请考虑一下，如果打算读取字节直到 readerIndex 达到和 writerIndex 同样的值时会发生什么。在那时，你将会到达“可以读取的”数据的末尾。就如同试图读取超出数组末尾的数据一样，试图读取超出该点的数据将会触发一个 IndexOutOfBoundsException。

名称以 read 或者 write 开头的 ByteBuf 方法，将会推进其对应的索引，而名称以 set 或者 get 开头的操作则不会。后面的这些方法将在作为一个参数传入的一个相对索引上执行操作。

可以指定 ByteBuf 的最大容量。试图移动写索引（即 writerIndex）超过这个值将会触发一个异常^①。（默认的限制是 Integer.MAX_VALUE。）

5.2.2 ByteBuf 的使用模式

在使用 Netty 时，你将遇到几种常见的围绕 ByteBuf 而构建的使用模式。在研究它们时，

^① 也就是说用户直接或者间接使 capacity(int) 或者 ensureWritable(int) 方法来增加超过该最大容量时抛出异常。——译者注

我们心里想着图5-1会有所裨益——一个由不同的索引分别控制读访问和写访问的字节数组。

1. 堆缓冲区

最常用的ByteBuf模式是将数据存储在JVM的堆空间中。这种模式被称为支撑数组(backing array)，它能在没有使用池化的情况下提供快速的分配和释放。这种方式，如代码清单5-1所示，非常适合于有遗留的数据需要处理的情况。

代码清单5-1 支撑数组

```
ByteBuf heapBuf = ...;
if (heapBuf.hasArray()) {
    byte[] array = heapBuf.array();
    int offset = heapBuf.arrayOffset() + heapBuf.readerIndex(); ← 检查 ByteBuf 是否
    int length = heapBuf.readableBytes(); ← 有一个支撑数组
    handleArray(array, offset, length); ← 如果有，则获取
} ← 对该数组的引用
    } ← 获得可读
    } ← 计算第一个字节
    } ← 字节数
    } ← 使用数组、偏移量和长度
    } ← 作为参数调用你的方法
```

注意 当hasArray()方法返回false时，尝试访问支撑数组将触发一个UnsupportedOperationException。这个模式类似于JDK的ByteBuffer的用法。

2. 直接缓冲区

直接缓冲区是另外一种ByteBuf模式。我们期望用于对象创建的内存分配永远都来自于堆中，但这并不是必须的——NIO在JDK 1.4中引入的ByteBuffer类允许JVM实现通过本地调用来分配内存。这主要是为了避免在每次调用本地I/O操作之前（或者之后）将缓冲区的内容复制到一个中间缓冲区（或者从中间缓冲区把内容复制到缓冲区）。

ByteBuffer的Javadoc^①明确指出：“直接缓冲区的内容将驻留在常规的会被垃圾回收的堆之外。”这也就解释了为何直接缓冲区对于网络数据传输是理想的选择。如果你的数据包含在一个在堆上分配的缓冲区中，那么事实上，在通过套接字发送它之前，JVM将会在内部把你的缓冲区复制到一个直接缓冲区中。

直接缓冲区的主要缺点是，相对于基于堆的缓冲区，它们的分配和释放都较为昂贵。如果你正在处理遗留代码，你也可能会遇到另外一个缺点：因为数据不是在堆上，所以你不得不进行一次复制，如代码清单5-2所示。

显然，与使用支撑数组相比，这涉及的工作更多。因此，如果事先知道容器中的数据将会被作为数组来访问，你可能更愿意使用堆内存。

^① Java平台，标准版第8版API规范，java.nio，class ByteBuffer：<http://docs.oracle.com/javase/8/docs/api/java/nio/ByteBuffer.html>。

代码清单 5-2 访问直接缓冲区的数据

```

ByteBuf directBuf = ...;
if (!directBuf.hasArray()) {
    int length = directBuf.readableBytes();
    byte[] array = new byte[length];
    directBuf.getBytes(directBuf.readerIndex(), array);
    handleArray(array, 0, length);
}

```

检查 ByteBuf 是否由数组支撑。如果不是，则这是一个直接缓冲区

获取可读字节数

分配一个新的数组来保存具有该长度的字节数据

使用数组、偏移量和长度作为参数调用你的方法

将字节复制到该数组

3. 复合缓冲区

第三种也是最后一种模式使用的是复合缓冲区，它为多个 ByteBuf 提供一个聚合视图。在这里你可以根据需要添加或者删除 ByteBuf 实例，这是一个 JDK 的 ByteBuffer 实现完全缺失的特性。

Netty 通过一个 ByteBuf 子类——CompositeByteBuf——实现了这个模式，它提供了一个将多个缓冲区表示为单个合并缓冲区的虚拟表示。

警告 CompositeByteBuf 中的 ByteBuf 实例可能同时包含直接内存分配和非直接内存分配。

如果其中只有一个实例，那么对 CompositeByteBuf 上的 hasArray() 方法的调用将返回该组件上的 hasArray() 方法的值；否则它将返回 false。

为了举例说明，让我们考虑一下一个由两部分——头部和主体——组成的将通过 HTTP 协议传输的消息。这两部分由应用程序的不同模块产生，将会在消息被发送的时候组装。该应用程序可以选择为多个消息重用相同的消息主体。当这种情况发生时，对于每个消息都将会创建一个新的头部。

因为我们不想为每个消息都重新分配这两个缓冲区，所以使用 CompositeByteBuf 是一个完美的选择。它在消除了没必要的复制的同时，暴露了通用的 ByteBuf API。图 5-2 展示了生成的消息布局。

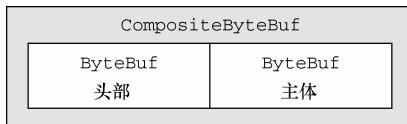


图 5-2 持有一个头部和主体的 CompositeByteBuf

代码清单 5-3 展示了如何通过使用 JDK 的 ByteBuffer 来实现这一需求。创建了一个包含两个 ByteBuffer 的数组用来保存这些消息组件，同时创建了第三个 ByteBuffer 用来保存所有这些数据的副本。

代码清单 5-3 使用 ByteBuffer 的复合缓冲区模式

```
// Use an array to hold the message parts
ByteBuffer[] message = new ByteBuffer[] { header, body };
// Create a new ByteBuffer and use copy to merge the header and body
ByteBuffer message2 =
    ByteBuffer.allocate(header.remaining() + body.remaining());
message2.put(header);
message2.put(body);
message2.flip();
```

分配和复制操作，以及伴随着对数组管理的需要，使得这个版本的实现效率低下而且笨拙。代码清单 5-4 展示了一个使用了 CompositeByteBuf 的版本。

代码清单 5-4 使用 CompositeByteBuf 的复合缓冲区模式

```
CompositeByteBuf messageBuf = Unpooled.compositeBuffer();
ByteBuf headerBuf = ...; // can be backing or direct
ByteBuf bodyBuf = ...; // can be backing or direct
messageBuf.addComponent(headerBuf, bodyBuf);
.....
messageBuf.removeComponent(0); // remove the header
for (ByteBuf buf : messageBuf) {
    System.out.println(buf.toString());
}
```

将 ByteBuffer 实例追加到 CompositeByteBuf

删除位于索引位置为 0 (第一个组件) 的 ByteBuffer

循环遍历所有 的 ByteBuffer 实例

CompositeByteBuf 可能不支持访问其支撑数组，因此访问 CompositeByteBuf 中的数据类似于（访问）直接缓冲区的模式，如代码清单 5-5 所示。

代码清单 5-5 访问 CompositeByteBuf 中的数据

```
CompositeByteBuf compBuf = Unpooled.compositeBuffer();
int length = compBuf.readableBytes();
byte[] array = new byte[length];
compBuf.getBytes(compBuf.readerIndex(), array);
handleArray(array, 0, array.length);
```

将字节读到 该数组中

获得可读字节数

分配一个具有可读字节数长度的新数组

使用偏移量和长度作为参数使用该数组

需要注意的是，Netty 使用了 CompositeByteBuf 来优化套接字的 I/O 操作，尽可能地消除了由 JDK 的缓冲区实现所导致的性能以及内存使用率的惩罚。^①这种优化发生在 Netty 的核心代码中，因此不会被暴露出来，但是你应该知道它所带来的影响。

^① 这尤其适用于 JDK 所使用的一种称为分散/收集 I/O (Scatter/Gather I/O) 的技术，定义为“一种输入和输出的方法，其中，单个系统调用从单个数据流写到一组缓冲区中，或者，从单个数据源读到一组缓冲区中”。《Linux System Programming》，作者 Robert Love (O'Reilly, 2007)。

CompositeByteBuf API 除了从 `ByteBuf` 继承的方法, `CompositeByteBuf` 提供了大量的附加功能。请参考 Netty 的 Javadoc 以获得该 API 的完整列表。

5.3 字节级操作

`ByteBuf` 提供了许多超出基本读、写操作的方法用于修改它的数据。在接下来的章节中, 我们将会讨论这些中最重要的一部分。

5.3.1 随机访问索引

如同在普通的 Java 字节数组中一样, `ByteBuf` 的索引是从零开始的: 第一个字节的索引是 0, 最后一个字节的索引总是 `capacity() - 1`。代码清单 5-6 表明, 对存储机制的封装使得遍历 `ByteBuf` 的内容非常简单。

代码清单 5-6 访问数据

```
ByteBuf buffer = ...;
for (int i = 0; i < buffer.capacity(); i++) {
    byte b = buffer.getByte(i);
    System.out.println((char)b);
}
```

需要注意的是, 使用那些需要一个索引值参数的方法 (的其中) 之一来访问数据既不会改变 `readerIndex` 也不会改变 `writerIndex`。如果有需要, 也可以通过调用 `readerIndex(index)` 或者 `writerIndex(index)` 来手动移动这两者。

5.3.2 顺序访问索引

虽然 `ByteBuf` 同时具有读索引和写索引, 但是 JDK 的 `ByteBuffer` 却只有一个索引, 这也就是为什么必须调用 `flip()` 方法 来在读模式和写模式之间进行切换的原因。图 5-3 展示了 `ByteBuf` 是如何被它的两个索引划分成 3 个区域的。

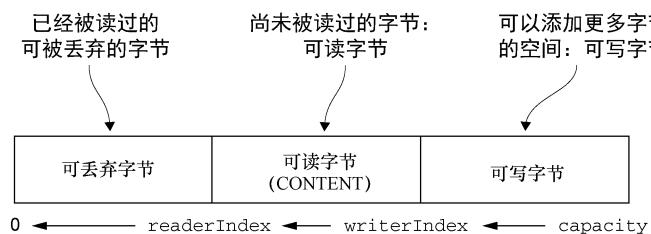


图 5-3 `ByteBuf` 的内部分段

5.3.3 可丢弃字节

在图 5-3 中标记为可丢弃字节的分段包含了已经被读过的字节。通过调用 `discardReadBytes()` 方法，可以丢弃它们并回收空间。这个分段的初始大小为 0，存储在 `readerIndex` 中，会随着 `read` 操作的执行而增加（`get*` 操作不会移动 `readerIndex`）。

图 5-4 展示了图 5-3 中所展示的缓冲区上调用 `discardReadBytes()` 方法后的结果。可以看到，可丢弃字节分段中的空间已经变为可写的了。注意，在调用 `discardReadBytes()` 之后，对可写分段的内容并没有任何的保证^①。

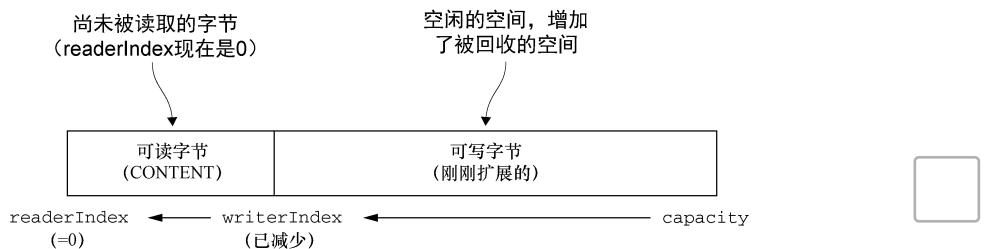


图 5-4 丢弃已读字节之后的 ByteBuf

虽然你可能会倾向于频繁地调用 `discardReadBytes()` 方法以确保可写分段的最大化，但是请注意，这将极有可能会导致内存复制，因为可读字节（图中标记为 `CONTENT` 的部分）必须被移动到缓冲区的开始位置。我们建议只在有真正需要的时候才这样做，例如，当内存非常宝贵的时候。

5.3.4 可读字节

ByteBuf 的可读字节分段存储了实际数据。新分配的、包装的或者复制的缓冲区的默认的 `readerIndex` 值为 0。任何名称以 `read` 或者 `skip` 开头的操作都将检索或者跳过位于当前 `readerIndex` 的数据，并且将它增加已读字节数。

如果被调用的方法需要一个 ByteBuf 参数作为写入的目标，并且没有指定目标索引参数，那么该目标缓冲区的 writerIndex 也将被增加，例如：

```
readBytes(ByteBuf dest);
```

如果尝试在缓冲区的可读字节数已经耗尽时从中读取数据，那么将会引发一个 `IndexOutOfBoundsException`。

代码清单 5-7 展示了如何读取所有可以读的数据。

代码清单 5-7 读取所有数据

```
ByteBuf buffer = ...;
while (buffer.isReadable()) {
```

^① 因为只是移动了可以读取的字节以及 writerIndex，而没有对所有可写入的字节进行擦除写。——译者注

```
System.out.println(buffer.readByte());
}
```

5.3.5 可写字符节

可写字符节分段是指一个拥有未定义内容的、写入就绪的内存区域。新分配的缓冲区的 writerIndex 的默认值为 0。任何名称以 write 开头的操作都将从当前的 writerIndex 处开始写数据，并将它增加已经写入的字节数。如果写操作的目标也是 ByteBuf，并且没有指定源索引的值，则源缓冲区的 readerIndex 也同样会被增加相同的大小。这个调用如下所示：

```
writeBytes(ByteBuf dest);
```

如果尝试往目标写入超过目标容量的数据，将会引发一个 IndexOutOfBoundsException^①。

代码清单 5-8 是一个用随机整数值填充缓冲区，直到它空间不足为止的例子。writeableBytes() 方法在这里被用来确定该缓冲区中是否还有足够的空间。

代码清单 5-8 写数据

```
// Fills the writable bytes of a buffer with random integers.
ByteBuf buffer = ...;
while (buffer.writableBytes() >= 4) {
    buffer.writeInt(random.nextInt());
}
```

5.3.6 索引管理

JDK 的 InputStream 定义了 mark(int readlimit) 和 reset() 方法，这些方法分别被用来将流中的当前位置标记为指定的值，以及将流重置到该位置。

同样，可以通过调用 markReaderIndex()、markWriterIndex()、resetWriterIndex() 和 resetReaderIndex() 来标记和重置 ByteBuf 的 readerIndex 和 writerIndex。这些和 InputStream 上的调用类似，只是没有 readlimit 参数来指定标记什么时候失效。

也可以通过调用 readerIndex(int) 或者 writerIndex(int) 来将索引移动到指定位置。试图将任何一个索引设置到一个无效的位置都将导致一个 IndexOutOfBoundsException。

可以通过调用 clear() 方法来将 readerIndex 和 writerIndex 都设置为 0。注意，这并不会清除内存中的内容。图 5-5（重复上面的图 5-3）展示了它是如何工作的。



图 5-5 clear() 方法被调用之前

^① 在往 ByteBuf 中写入数据时，其将首先确保目标 ByteBuf 具有足够的可写入空间来容纳当前要写入的数据，如果没有，则将检查当前的写索引以及最大容量是否可以在扩展后容纳该数据，可以则会分配并调整容量，否则就会抛出该异常。——译者注

和之前一样，ByteBuf 包含 3 个分段。图 5-6 展示了在 clear() 方法被调用之后 ByteBuf 的状态。

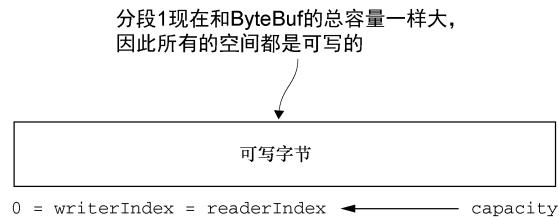


图 5-6 在 clear() 方法被调用之后

调用 clear() 比调用 discardReadBytes() 轻量得多，因为它将只是重置索引而不会复制任何的内存。

5.3.7 查找操作

在 ByteBuf 中有多种可以用来确定指定值的索引的方法。最简单的是使用 indexOf() 方法。较复杂的查找可以通过那些需要一个 ByteBufProcessor^① 作为参数的方法达成。这个接口只定义了一个方法：

```
boolean process(byte value)
```

它将检查输入值是否是正在查找的值。

ByteBufProcessor 针对一些常见的值定义了许多便利的方法。假设你的应用程序需要和所谓的包含有以 NULL 结尾的内容的 Flash 套接字^②集成。调用

```
forEachByte(ByteBufProcessor.FIND_NUL)
```

将简单高效地消费该 Flash 数据，因为在处理期间只会执行较少的边界检查。

代码清单 5-9 展示了一个找回车符 (\r) 的例子。

代码清单 5-9 使用 ByteBufProcessor 来寻找\r

```
ByteBuf buffer = ...;
int index = buffer.forEachByte(ByteBufProcessor.FIND_CR);
```

5.3.8 派生缓冲区

派生缓冲区为 ByteBuf 提供了以专门的方式来呈现其内容的视图。这类视图是通过以下方法被创建的：

- `duplicate()`;
- `slice()`;

① 在 Netty 4.1.x 中，该类已经废弃，请使用 `io.netty.util.ByteProcessor`。——译者注

② 有关 Flash 套接字的讨论可参考 Flash ActionScript 3.0 Developer's Guide 中 Networking and Communication 部分里的 Sockets 页面：http://help.adobe.com/en_US/as3/dev/WSb2ba3b1aad8a27b0-181c51321220efd9d1c-8000.html。

- slice(int, int);
- Unpooled.unmodifiableBuffer(...);
- order(ByteOrder);
- readSlice(int)。

每个这些方法都将返回一个新的 ByteBuf 实例，它具有自己的读索引、写索引和标记索引。其内部存储和 JDK 的 ByteBuffer 一样也是共享的。这使得派生缓冲区的创建成本是很低廉的，但是这也意味着，如果你修改了它的内容，也同时修改了其对应的源实例，所以要小心。

ByteBuf 复制 如果需要一个现有缓冲区的真实副本，请使用 copy() 或者 copy(int, int) 方法。不同于派生缓冲区，由这个调用所返回的 ByteBuf 拥有独立的数据副本。

代码清单 5-10 展示了如何使用 slice(int, int) 方法来操作 ByteBuf 的一个分段。

代码清单 5-10 对 ByteBuf 进行切片

```
创建一个用于保存给定字符串
串的字节的 ByteBuf
Charset utf8 = Charset.forName("UTF-8");
→ ByteBuf buf = Unpooled.copiedBuffer("Netty in Action rocks!", utf8);
ByteBuf sliced = buf.slice(0, 15);
System.out.println(sliced.toString(utf8));
buf.setByte(0, (byte)'J');
assert buf.getByte(0) == sliced.getByte(0);
```

创建该 ByteBuf 从索引 0 开始到索引 15 结束的一个新切片

将打印 "Netty in Action"

更新索引 0 处的字节

将会成功，因为数据是共享的，对其中一个所做的更改对另外一个也是可见的

现在，让我们看看 ByteBuf 的分段的副本和切片有何区别，如代码清单 5-11 所示。

代码清单 5-11 复制一个 ByteBuf

```
创建 ByteBuf 以保存
所提供的字符串的字节
Charset utf8 = Charset.forName("UTF-8");
→ ByteBuf buf = Unpooled.copiedBuffer("Netty in Action rocks!", utf8);
ByteBuf copy = buf.copy(0, 15);
System.out.println(copy.toString(utf8));
buf.setByte(0, (byte)'J');
assert buf.getByte(0) != copy.getByte(0);
```

创建该 ByteBuf 从索引 0 开始到索引 15 结束的分段的副本

将打印 "Netty in Action"

更新索引 0 处的字节

将会成功，因为数据不是共享的

除了修改原始 ByteBuf 的切片或者副本的效果以外，这两种场景是相同的。只要有可能，使用 slice() 方法来避免复制内存的开销。

5.3.9 读/写操作

正如我们所提到过的，有两种类别的读/写操作：

- `get()` 和 `set()` 操作，从给定的索引开始，并且保持索引不变；
- `read()` 和 `write()` 操作，从给定的索引开始，并且会根据已经访问过的字节数对索引进行调整。

表 5-1 列举了最常用的 `get()` 方法。完整列表请参考对应的 API 文档。

表 5-1 `get()` 操作

名 称	描 述
<code>getBoolean(int)</code>	返回给定索引处的 Boolean 值
<code>getByte(int)</code>	返回给定索引处的字节
<code>getUnsignedByte(int)</code>	将给定索引处的无符号字节值作为 short 返回
<code>getMedium(int)</code>	返回给定索引处的 24 位的中等 int 值
<code>getUnsignedMedium(int)</code>	返回给定索引处的无符号的 24 位的中等 int 值
<code>getInt(int)</code>	返回给定索引处的 int 值
<code>getUnsignedInt(int)</code>	将给定索引处的无符号 int 值作为 long 返回
<code>getLong(int)</code>	返回给定索引处的 long 值
<code>getShort(int)</code>	返回给定索引处的 short 值
<code>getUnsignedShort(int)</code>	将给定索引处的无符号 short 值作为 int 返回
<code>getBytes(int, ...)</code>	将该缓冲区中从给定索引开始的数据传送到指定的目的地

大多数的这些操作都有一个对应的 `set()` 方法。这些方法在表 5-2 中列出。

表 5-2 `set()` 操作

名 称	描 述
<code>setBoolean(int, boolean)</code>	设定给定索引处的 Boolean 值
<code>setByte(int index, int value)</code>	设定给定索引处的字节值
<code>setMedium(int index, int value)</code>	设定给定索引处的 24 位的中等 int 值
<code>setInt(int index, int value)</code>	设定给定索引处的 int 值
<code>setLong(int index, long value)</code>	设定给定索引处的 long 值
<code>setShort(int index, int value)</code>	设定给定索引处的 short 值

代码清单 5-12 说明了 `get()` 和 `set()` 方法的用法，表明了它们不会改变读索引和写索引。

代码清单 5-12 `get()` 和 `set()` 方法的用法

```

创建一个新的 ByteBuf
以保存给定字符串的字节
Charset utf8 = Charset.forName("UTF-8");
ByteBuf buf = Unpooled.copiedBuffer("Netty in Action rocks!", utf8);
System.out.println((char)buf.getByte(0));
int readerIndex = buf.readerIndex();           ↗ 存储当前的 readerIndex 和 writerIndex
int writerIndex = buf.writerIndex();
buf.setByte(0, (byte)'B');
System.out.println((char)buf.getByte(0));
assert readerIndex == buf.readerIndex();       ↗ 将索引 0 处的字
assert writerIndex == buf.writerIndex();        ↗ 节更新为字符'B'
                                                     ↗ 打印第一个字
                                                     符, 现在是'B'
                                                     ↗ 打印第一个字
                                                     符, 现在是'B'

将会成功, 因为这些操作
并不会修改相应的索引

```

现在, 让我们研究一下 `read()` 操作, 其作用于当前的 `readerIndex` 或 `writerIndex`。这些方法将用于从 `ByteBuf` 中读取数据, 如同它是一个流。表 5-3 展示了最常用的方法。

表 5-3 `read()` 操作

名 称	描 述
<code>readBoolean()</code>	返回当前 <code>readerIndex</code> 处的 Boolean, 并将 <code>readerIndex</code> 增加 1
<code>readByte()</code>	返回当前 <code>readerIndex</code> 处的字节, 并将 <code>readerIndex</code> 增加 1
<code>readUnsignedByte()</code>	将当前 <code>readerIndex</code> 处的无符号字节值作为 <code>short</code> 返回, 并将 <code>readerIndex</code> 增加 1
<code>readMedium()</code>	返回当前 <code>readerIndex</code> 处的 24 位的中等 <code>int</code> 值, 并将 <code>readerIndex</code> 增加 3
<code>readUnsignedMedium()</code>	返回当前 <code>readerIndex</code> 处的 24 位的无符号的中等 <code>int</code> 值, 并将 <code>readerIndex</code> 增加 3
<code>readInt()</code>	返回当前 <code>readerIndex</code> 的 <code>int</code> 值, 并将 <code>readerIndex</code> 增加 4
<code>readUnsignedInt()</code>	将当前 <code>readerIndex</code> 处的无符号的 <code>int</code> 值作为 <code>long</code> 值返回, 并将 <code>readerIndex</code> 增加 4
<code>readLong()</code>	返回当前 <code>readerIndex</code> 处的 <code>long</code> 值, 并将 <code>readerIndex</code> 增加 8
<code>readShort()</code>	返回当前 <code>readerIndex</code> 处的 <code>short</code> 值, 并将 <code>readerIndex</code> 增加 2
<code>readUnsignedShort()</code>	将当前 <code>readerIndex</code> 处的无符号 <code>short</code> 值作为 <code>int</code> 值返回, 并将 <code>readerIndex</code> 增加 2
<code>readBytes(ByteBuf byte[] destination, int dstIndex [,int length])</code>	将当前 <code>ByteBuf</code> 中从当前 <code>readerIndex</code> 处开始的 (如果设置了, <code>length</code> 长度的字节) 数据传送到一个目标 <code>ByteBuf</code> 或者 <code>byte[]</code> , 从目标的 <code>dstIndex</code> 开始的位置。本地的 <code>readerIndex</code> 将被增加已经传输的字节数

几乎每个 `read()` 方法都有对应的 `write()` 方法，用于将数据追加到 `ByteBuf` 中。注意，表 5-4 中所列出的这些方法的参数是需要写入的值，而不是索引值。

表 5-4 写操作

名 称	描 述
<code>writeBoolean(boolean)</code>	在当前 <code>writerIndex</code> 处写入一个 <code>Boolean</code> ，并将 <code>writerIndex</code> 增加 1
<code>writeByte(int)</code>	在当前 <code>writerIndex</code> 处写入一个字节值，并将 <code>writerIndex</code> 增加 1
<code>writeMedium(int)</code>	在当前 <code>writerIndex</code> 处写入一个中等的 <code>int</code> 值，并将 <code>writerIndex</code> 增加 3
<code>writeInt(int)</code>	在当前 <code>writerIndex</code> 处写入一个 <code>int</code> 值，并将 <code>writerIndex</code> 增加 4
<code>writeLong(long)</code>	在当前 <code>writerIndex</code> 处写入一个 <code>long</code> 值，并将 <code>writerIndex</code> 增加 8
<code>writeShort(int)</code>	在当前 <code>writerIndex</code> 处写入一个 <code>short</code> 值，并将 <code>writerIndex</code> 增加 2
<code>writeBytes(source ByteBuf byte[] [, int srcIndex, int length])</code>	从当前 <code>writerIndex</code> 开始，传输来自于指定源(<code>ByteBuf</code> 或者 <code>byte[]</code>)的数据。如果提供了 <code>srcIndex</code> 和 <code>length</code> ，则从 <code>srcIndex</code> 开始读取，并且处理长度为 <code>length</code> 的字节。当前 <code>writerIndex</code> 将会被增加所写入的字节数

代码清单 5-13 展示了这些方法的用法。

代码清单 5-13 ByteBuf 上的 `read()` 和 `write()` 操作

```
创建一个新的 ByteBuf 以保存
给定字符串的字节
Charset utf8 = Charset.forName("UTF-8");
ByteBuf buf = Unpooled.copiedBuffer("Netty in Action rocks!", utf8);
System.out.println((char)buf.readByte());           // 打印第一个字符'N'
int readerIndex = buf.readerIndex();                // 存储当前的 readerIndex
int writerIndex = buf.writerIndex();                 // 存储当前的 writerIndex
buf.writeByte((byte)'?');                           // 将字符'?'追加到缓冲区
assert readerIndex == buf.readerIndex();            // 将会成功，因为 writeByte() 方法移动了 writerIndex
assert writerIndex != buf.writerIndex();
```

5.3.10 更多的操作

表 5-5 列举了由 `ByteBuf` 提供的其他有用操作。

表 5-5 其他有用的操作

名 称	描 述
<code>isReadable()</code>	如果至少有一个字节可供读取，则返回 <code>true</code>
<code>isWritable()</code>	如果至少有一个字节可被写入，则返回 <code>true</code>

续表

名 称	描 述
readableBytes()	返回可被读取的字节数
writableBytes()	返回可被写入的字节数
capacity()	返回 ByteBuf 可容纳的字节数。在此之后，它会尝试再次扩展直到达到 maxCapacity()
maxCapacity()	返回 ByteBuf 可以容纳的最大字节数
hasArray()	如果 ByteBuf 由一个字节数组支撑，则返回 true
array()	如果 ByteBuf 由一个字节数组支撑则返回该数组；否则，它将抛出一个 UnsupportedOperationException 异常

5.4 ByteBufHolder 接口

我们经常发现，除了实际的数据负载之外，我们还需要存储各种属性值。HTTP 响应便是一个很好的例子，除了表示为字节的内容，还包括状态码、cookie 等。

为了处理这种常见的用例，Netty 提供了 ByteBufHolder。ByteBufHolder 也为 Netty 的高级特性提供了支持，如缓冲区池化，其中可以从池中借用 ByteBuf，并且在需要时自动释放。

ByteBufHolder 只有几种用于访问底层数据和引用计数的方法。表 5-6 列出了它们（这里不包括它继承自 ReferenceCounted 的那些方法）。

表 5-6 ByteBufHolder 的操作

名 称	描 述
content()	返回由这个 ByteBufHolder 所持有的 ByteBuf
copy()	返回这个 ByteBufHolder 的一个深拷贝，包括一个其所包含的 ByteBuf 的非共享拷贝
duplicate()	返回这个 ByteBufHolder 的一个浅拷贝，包括一个其所包含的 ByteBuf 的共享拷贝

如果想要实现一个将其有效负载存储在 ByteBuf 中的消息对象，那么 ByteBufHolder 将是个不错的选择。

5.5 ByteBuf 分配

在这一节中，我们将描述管理 ByteBuf 实例的不同方式。

5.5.1 按需分配：ByteBufAllocator 接口

为了降低分配和释放内存的开销，Netty 通过 interface ByteBufAllocator 实现了（ByteBuf 的）池化，它可以用来分配我们所描述过的任意类型的 ByteBuf 实例。使用池化是

特定于应用程序的决定，其并不会以任何方式改变 ByteBuf API (的语义)。

表 5-7 列出了 ByteBufAllocator 提供的一些操作。

表 5-7 ByteBufAllocator 的方法

名 称	描 述
buffer() buffer(int initialCapacity); buffer(int initialCapacity, int maxCapacity);	返回一个基于堆或者直接内存存储的 ByteBuf
heapBuffer() heapBuffer(int initialCapacity) heapBuffer(int initialCapacity, int maxCapacity)	返回一个基于堆内存存储的 ByteBuf
directBuffer() directBuffer(int initialCapacity) directBuffer(int initialCapacity, int maxCapacity)	返回一个基于直接内存存储的 ByteBuf
compositeBuffer() compositeBuffer(int maxNumComponents) compositeDirectBuffer() compositeDirectBuffer(int maxNumComponents); compositeHeapBuffer() compositeHeapBuffer(int maxNumComponents);	返回一个可以通过添加最大到指定数目的基于堆的或者直接内存存储的缓冲区来扩展的 CompositeByteBuf
ioBuffer() ^①	返回一个用于套接字的 I/O 操作的 ByteBuf

可以通过 Channel (每个都可以有一个不同的 ByteBufAllocator 实例) 或者绑定到 ChannelHandler 的 ChannelHandlerContext 获取一个到 ByteBufAllocator 的引用。代码清单 5-14 说明了这两种方法。

代码清单 5-14 获取一个到 ByteBufAllocator 的引用

```
Channel channel = ...;
ByteBufAllocator allocator = channel.alloc();           ← 从 Channel 获取一个到
...                                         ByteBufAllocator 的引用
ChannelHandlerContext ctx = ...;
ByteBufAllocator allocator2 = ctx.alloc();   ← 从 ChannelHandlerContext 获取一个
...                                         到 ByteBufAllocator 的引用
```

Netty 提供了两种 ByteBufAllocator 的实现：PooledByteBufAllocator 和 Unpooled-ByteBufAllocator。前者池化了 ByteBuf 的实例以提高性能并最大限度地减少内存碎片。此实现使用了一种称为 jemalloc^② 的已被大量现代操作系统所采用的高效方法来分配内存。后者的实现不

① 默认地，当所运行的环境具有 sun.misc.Unsafe 支持时，返回基于直接内存存储的 ByteBuf，否则返回基于堆内存存储的 ByteBuf；当指定使用 PreferHeapByteBufAllocator 时，则只会返回基于堆内存存储的 ByteBuf。——译者注

② Jason Evans 的“ A Scalable Concurrent malloc(3) Implementation for FreeBSD”(2006): <http://people.freebsd.org/~jasone/jemalloc/bsdcan2006/jemalloc.pdf>。

池化ByteBuf实例，并且在每次它被调用时都会返回一个新的实例。

虽然Netty默认^①使用了PooledByteBufAllocator，但这可以很容易地通过ChannelConfig API或者在引导你的应用程序时指定一个不同的分配器来更改。更多的细节可在第8章中找到。

5.5.2 Unpooled 缓冲区

可能某些情况下，你未能获取一个到ByteBufAllocator的引用。对于这种情况，Netty提供了一个简单的称为Unpooled的工具类，它提供了静态的辅助方法来创建未池化的ByteBuf实例。表5-8列举了这些中最重要的方法。

表5-8 Unpooled的方法

名 称	描 述
buffer() buffer(int initialCapacity) buffer(int initialCapacity, int maxCapacity)	返回一个未池化的基于堆内存存储的ByteBuf
directBuffer() directBuffer(int initialCapacity) directBuffer(int initialCapacity, int maxCapacity)	返回一个未池化的基于直接内存存储的ByteBuf
wrappedBuffer()	返回一个包装了给定数据的ByteBuf
copiedBuffer()	返回一个复制了给定数据的ByteBuf

Unpooled类还使得ByteBuf同样可用于那些并不需要Netty的其他组件的非网络项目，使得其能得益于高性能的可扩展的缓冲区API。

5.5.3 ByteBufUtil类

ByteBufUtil提供了用于操作ByteBuf的静态的辅助方法。因为这个API是通用的，并且和池化无关，所以这些方法已然在分配类的外部实现。

这些静态方法中最有价值的可能就是hexdump()方法，它以十六进制的表示形式打印ByteBuf的内容。这在各种情况下都很有用，例如，出于调试的目的记录ByteBuf的内容。十六进制的表示通常会提供一个比字节值的直接表示形式更加有用的日志条目，此外，十六进制的版本还可以很容易地转换回实际的字节表示。

另一个有用的方法是boolean equals(ByteBuf, ByteBuf)，它被用来判断两个ByteBuf实例的相等性。如果你实现自己的ByteBuf子类，你可能会发现ByteBufUtil的其他有用方法。

5.6 引用计数

引用计数是一种通过在某个对象所持有的资源不再被其他对象引用时释放该对象所持有的

① 这里指Netty4.1.x，Netty4.0.x默认使用的是UnpooledByteBufAllocator。——译者注

资源来优化内存使用和性能的技术。Netty 在第 4 版中为 `ByteBuf` 和 `ByteBufHolder` 引入了引用计数技术，它们都实现了 `interface ReferenceCounted`。

引用计数背后的想法并不是特别的复杂；它主要涉及跟踪到某个特定对象的活动引用的数量。一个 `ReferenceCounted` 实现的实例将通常以活动的引用计数为 1 作为开始。只要引用计数大于 0，就能保证对象不会被释放。当活动引用的数量减少到 0 时，该实例就会被释放。注意，虽然释放的确切语义可能是特定于实现的，但是至少已经释放的对象应该不可再用了。

引用计数对于池化实现（如 `PooledByteBufAllocator`）来说是至关重要的，它降低了内存分配的开销。代码清单 5-15 和代码清单 5-16 展示了相关的示例。

代码清单 5-15 引用计数

```
Channel channel = ...;
ByteBufAllocator allocator = channel.alloc();
...
ByteBuf buffer = allocator.directBuffer();
assert buffer.refCnt() == 1;
...
    
```

The diagram shows the code from the listing. Annotations explain the process: '从 Channel 获取 ByteBufAllocator' (Get ByteBufAllocator from Channel) points to the line `ByteBufAllocator allocator = channel.alloc();`; '从 ByteBufAllocator 分配一个 ByteBuf' (Allocate a ByteBuf from ByteBufAllocator) points to the line `ByteBuf buffer = allocator.directBuffer();`; and '检查引用计数是否为预期的 1' (Check if reference count is as expected) points to the line `assert buffer.refCnt() == 1;`.

代码清单 5-16 释放引用计数的对象

```
ByteBuf buffer = ...;
boolean released = buffer.release();
...
    
```

The diagram shows the code from the listing. An annotation '减少到该对象的活动引用。当减少到 0 时，该对象被释放，并且该方法返回 true' (Decrease the active reference count of the object. When it reaches 0, the object is released and the method returns true) points to the line `boolean released = buffer.release();`.

试图访问一个已经被释放的引用计数的对象，将会导致一个 `IllegalReferenceCountException`。

注意，一个特定的（`ReferenceCounted` 的实现）类，可以用它自己的独特方式来定义它的引用计数规则。例如，我们可以设想一个类，其 `release()` 方法的实现总是将引用计数设为零，而不用关心它的当前值，从而一次性地使所有的活动引用都失效。

谁负责释放 一般来说，是由最后访问（引用计数）对象的那一方来负责将它释放。在第 6 章中，我们将会解释这个概念和 `ChannelHandler` 以及 `ChannelPipeline` 的相关性。

5.7 小结

本章专门探讨了 Netty 的基于 `ByteBuf` 的数据容器。我们首先解释了 `ByteBuf` 相对于 JDK 所提供的实现的优势。我们还强调了该 API 的其他可用变体，并且指出了它们各自最佳适用的特定用例。

我们讨论过的要点有：

- 使用不同的读索引和写索引来控制数据访问；

- 使用内存的不同方式——基于字节数组和直接缓冲区；
- 通过 CompositeByteBuf 生成多个 ByteBuf 的聚合视图；
- 数据访问方法——搜索、切片以及复制；
- 读、写、获取和设置 API；
- ByteBufAllocator 池化和引用计数。

在下一章中，我们将专注于 ChannelHandler，它为你的数据处理逻辑提供了载体。因为 ChannelHandler 大量地使用了 ByteBuf，你将开始看到 Netty 的整体架构的各个重要部分最终走到了一起。

第 6 章 ChannelHandler 和 ChannelPipeline

本章主要内容

- ChannelHandler API 和 ChannelPipeline API
- 检测资源泄漏
- 异常处理

在上一章中你学习了 ByteBuf——Netty 的数据容器。当我们在本章中探讨 Netty 的数据流以及处理组件时，我们将基于已经学过的东西，并且你将开始看到框架的重要元素都结合到了一起。

你已经知道，可以在 ChannelPipeline 中将 ChannelHandler 链接在一起以组织处理逻辑。我们将会研究涉及这些类的各种用例，以及一个重要的关系——ChannelHandlerContext。

理解所有这些组件之间的交互对于通过 Netty 构建模块化的、可重用的实现至关重要。

6.1 ChannelHandler 家族

在我们开始详细地学习 ChannelHandler 之前，我们将在 Netty 的组件模型的这部分基础上花上一些时间。

6.1.1 Channel 的生命周期

Interface Channel 定义了一组和 ChannelInboundHandler API 密切相关的简单但功能强大的状态模型，表 6-1 列出了 Channel 的这 4 个状态。

表 6-1 Channel 的生命周期状态

状 态	描 述
ChannelUnregistered	Channel 已经被创建，但还未注册到 EventLoop
ChannelRegistered	Channel 已经被注册到了 EventLoop

续表

状态	描述
ChannelActive	Channel 处于活动状态(已经连接到它的远程节点)。它现在可以接收和发送数据了。
ChannelInactive	Channel 没有连接到远程节点。

Channel 的正常生命周期如图 6-1 所示。当这些状态发生改变时，将会生成对应的事件。这些事件将会被转发给 ChannelPipeline 中的 ChannelHandler，其可以随后对它们做出响应。

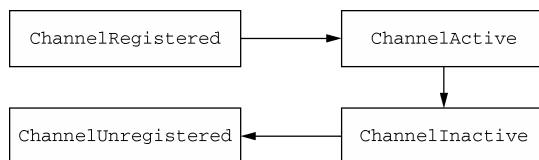


图 6-1 Channel 的状态模型

6.1.2 ChannelHandler 的生命周期

表 6-2 中列出了 interface ChannelHandler 定义的生命周期操作，在 ChannelHandler 被添加到 ChannelPipeline 中或者被从 ChannelPipeline 中移除时会调用这些操作。这些方法中的每一个都接受一个 ChannelHandlerContext 参数。

表 6-2 ChannelHandler 的生命周期方法

类型	描述
handlerAdded	当把 ChannelHandler 添加到 ChannelPipeline 中时被调用
handlerRemoved	当从 ChannelPipeline 中移除 ChannelHandler 时被调用
exceptionCaught	当处理过程中在 ChannelPipeline 中有错误产生时被调用

Netty 定义了下面两个重要的 ChannelHandler 子接口：

- ChannelInboundHandler——处理入站数据以及各种状态变化；
- ChannelOutboundHandler——处理出站数据并且允许拦截所有的操作。

在接下来的章节中，我们将详细地讨论这些子接口。

6.1.3 ChannelInboundHandler 接口

表 6-3 列出了 interface ChannelInboundHandler 的生命周期方法。这些方法将会在数据被接收时或者与其对应的 Channel 状态发生改变时被调用。正如我们前面所提到的，这些方法和 Channel 的生命周期密切相关。

表 6-3 ChannelInboundHandler 的方法

类 型	描 述
channelRegistered	当 Channel 已经注册到它的 EventLoop 并且能够处理 I/O 时被调用
channelUnregistered	当 Channel 从它的 EventLoop 注销并且无法处理任何 I/O 时被调用
channelActive	当 Channel 处于活动状态时被调用; Channel 已经连接/绑定并且已经就绪
channelInactive	当 Channel 离开活动状态并且不再连接它的远程节点时被调用
channelReadComplete	当 Channel 上的一个读操作完成时被调用 ^①
channelRead	当从 Channel 读取数据时被调用
ChannelWritability-Changed	当 Channel 的可写状态发生改变时被调用。用户可以确保写操作不会完成得太快 (以避免发生 OutOfMemoryError) 或者可以在 Channel 变为再次可写时恢复写入。可以通过调用 Channel 的 isWritable() 方法来检测 Channel 的可写性。与可写性相关的阈值可以通过 Channel.config().setWriteHighWaterMark() 和 Channel.config().setWriteLowWaterMark() 方法来设置
userEventTriggered	当 ChannelInboundHandler.fireUserEventTriggered() 方法被调用时被调用, 因为一个 POJO 被传经了 ChannelPipeline

当某个 ChannelInboundHandler 的实现重写 channelRead() 方法时, 它将负责显式地释放与池化的 ByteBuf 实例相关的内存。Netty 为此提供了一个实用方法 ReferenceCountUtil.release(), 如代码清单 6-1 所示。

代码清单 6-1 释放消息资源

```
@Sharable
public class DiscardHandler extends ChannelInboundHandlerAdapter {
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) {
        ReferenceCountUtil.release(msg);
    }
}
```

Netty 将使用 WARN 级别的日志消息记录未释放的资源, 使得可以非常简单地在代码中发现违规的实例。但是以这种方式管理资源可能很繁琐。一个更加简单的方式是使用 SimpleChannelInboundHandler。代码清单 6-2 是代码清单 6-1 的一个变体, 说明了这一点。

代码清单 6-2 使用 SimpleChannelInboundHandler

```
@Sharable
```

^① 当所有可读的字节都已经从 Channel 中读取之后, 将会调用该回调方法; 所以, 可能在 channelReadComplete() 被调用之前看到多次调用 channelRead(...). ——译者注

```

public class SimpleDiscardHandler
    extends SimpleChannelInboundHandler<Object> { ← 扩展了
    @Override
    public void channelRead0(ChannelHandlerContext ctx,
        Object msg) {
        // No need to do anything special ← 不需要任何显
    }
}                                式的资源释放

```

由于 `SimpleChannelInboundHandler` 会自动释放资源，所以你不应该存储指向任何消息的引用供将来使用，因为这些引用都将会失效。

6.1.6 节为引用处理提供了更加详细的讨论。

6.1.4 ChannelOutboundHandler 接口

出站操作和数据将由 `ChannelOutboundHandler` 处理。它的方法将被 `Channel`、`ChannelPipeline` 以及 `ChannelHandlerContext` 调用。

`ChannelOutboundHandler` 的一个强大的功能是可以按需推迟操作或者事件，这使得可以通过一些复杂的方法来处理请求。例如，如果到远程节点的写入被暂停了，那么你可以推迟冲刷操作并在稍后继续。

表 6-4 显示了所有由 `ChannelOutboundHandler` 本身所定义的方法(忽略了那些从 `ChannelHandler` 继承的方法)。

表 6-4 `ChannelOutboundHandler` 的方法

类 型	描 述
<code>bind(ChannelHandlerContext, SocketAddress, ChannelPromise)</code>	当请求将 Channel 绑定到本地地址时被调用
<code>connect(ChannelHandlerContext, SocketAddress, SocketAddress, ChannelPromise)</code>	当请求将 Channel 连接到远程节点时被调用
<code>disconnect(ChannelHandlerContext, ChannelPromise)</code>	当请求将 Channel 从远程节点断开时被调用
<code>close(ChannelHandlerContext, ChannelPromise)</code>	当请求关闭 Channel 时被调用
<code>deregister(ChannelHandlerContext, ChannelPromise)</code>	当请求将 Channel 从它的 EventLoop 注销时被调用
<code>read(ChannelHandlerContext)</code>	当请求从 Channel 读取更多的数据时被调用
<code>flush(ChannelHandlerContext)</code>	当请求通过 Channel 将入队数据冲刷到远程节点时被调用
<code>write(ChannelHandlerContext, Object, ChannelPromise)</code>	当请求通过 Channel 将数据写到远程节点时被调用

ChannelPromise与ChannelFuture ChannelOutboundHandler中的大部分方法都需要一个 ChannelPromise参数，以便在操作完成时得到通知。ChannelPromise是ChannelFuture的一个子类，其定义了一些可写的方法，如setSuccess()和setFailure()，从而使ChannelFuture不可变^①。

接下来我们将看一看那些简化了编写 ChannelHandler 的任务的类。

6.1.5 ChannelHandler 适配器

你可以使用 ChannelInboundHandlerAdapter 和 ChannelOutboundHandlerAdapter 类作为自己的 ChannelHandler 的起始点。这两个适配器分别提供了 ChannelInboundHandler 和 ChannelOutboundHandler 的基本实现。通过扩展抽象类 ChannelHandlerAdapter，它们获得了它们共同的超接口 ChannelHandler 的方法。生成的类的层次结构如图 6-2 所示。

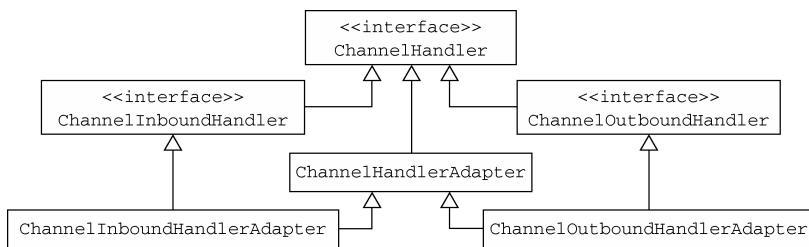


图 6-2 ChannelHandlerAdapter 类的层次结构

ChannelHandlerAdapter 还提供了实用方法 `isSharable()`。如果其对应的实现被标注为 Sharable，那么这个方法将返回 `true`，表示它可以被添加到多个 ChannelPipeline 中（如在 2.3.1 节中所讨论过的一样）。

在 ChannelInboundHandlerAdapter 和 ChannelOutboundHandlerAdapter 中所提供的方法体调用了其相关联的 ChannelHandlerContext 上的等效方法，从而将事件转发到了 ChannelPipeline 中的下一个 ChannelHandler 中。

你要想在自己的 ChannelHandler 中使用这些适配器类，只需要简单地扩展它们，并且重写那些你想要自定义的方法。

6.1.6 资源管理

每当通过调用 `ChannelInboundHandler.channelRead()` 或者 `ChannelOutboundHandler.write()` 方法来处理数据时，你都需要确保没有任何的资源泄漏。你可能还记得在前

^① 这里借鉴的是 Scala 的 Promise 和 Future 的设计，当一个 Promise 被完成之后，其对应的 Future 的值便不能再进行任何修改了。——译者注

面的章节中所提到的，Netty 使用引用计数来处理池化的 ByteBuf。所以在完全使用完某个 ByteBuf 后，调整其引用计数是很重要的。

为了帮助你诊断潜在的（资源泄漏）问题，Netty 提供了 class ResourceLeakDetector^①，它将对你应用程序的缓冲区分配做大约 1% 的采样来检测内存泄露。相关的开销是非常小的。

如果检测到了内存泄露，将会产生类似于下面的日志消息：

```
LEAK: ByteBuf.release() was not called before it's garbage-collected. Enable
advanced leak reporting to find out where the leak occurred. To enable
advanced leak reporting, specify the JVM option
'-Dio.netty.leakDetectionLevel=ADVANCED' or call
ResourceLeakDetector.setLevel().
```

Netty 目前定义了 4 种泄漏检测级别，如表 6-5 所示。

表 6-5 泄漏检测级别

级 别	描 述
DISABLED	禁用泄漏检测。只有在详尽的测试之后才应设置为这个值
SIMPLE	使用 1% 的默认采样率检测并报告任何发现的泄露。这是默认级别，适合绝大部分的情况
ADVANCED	使用默认的采样率，报告所发现的任何的泄露以及对应的消息被访问的位置
PARANOID	类似于 ADVANCED，但是其将会对每次（对消息的）访问都进行采样。这对性能将会有很大的影响，应该只在调试阶段使用

泄露检测级别可以通过将下面的 Java 系统属性设置为表中的一个值来定义：

```
java -Dio.netty.leakDetectionLevel=ADVANCED
```

如果带着该 JVM 选项重新启动你的应用程序，你将看到自己的应用程序最近被泄漏的缓冲区被访问的位置。下面是一个典型的由单元测试产生的泄露报告：

```
Running io.netty.handler.codec.xml.XmlFrameDecoderTest
15:03:36.886 [main] ERROR io.netty.util.ResourceLeakDetector - LEAK:
    ByteBuf.release() was not called before it's garbage-collected.
Recent access records: 1
#1: io.netty.buffer.AdvancedLeakAwareByteBuf.toString(
    AdvancedLeakAwareByteBuf.java:697)
io.netty.handler.codec.xml.XmlFrameDecoderTest.testDecodeWithXml(
    XmlFrameDecoderTest.java:157)
io.netty.handler.codec.xml.XmlFrameDecoderTest.testDecodeWithTwoMessages(
    XmlFrameDecoderTest.java:133)
...
```

实现 ChannelInboundHandler.channelRead() 和 ChannelOutboundHandler.write() 方法时，应该如何使用这个诊断工具来防止泄露呢？让我们看看你的 channelRead() 操作直接消费入站消息的情况；也就是说，它不会通过调用 ChannelHandlerContext.fireChannelRead() 方法将入站消息转发给下一个 ChannelInboundHandler。代码清单 6-3 展示了如何释放消息。

① 其利用了 JDK 提供的 PhantomReference<T>类来实现这一点。——译者注

代码清单 6-3 消费并释放入站消息

```

@Sharable
public class DiscardInboundHandler extends ChannelInboundHandlerAdapter {
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) {
        ReferenceCountUtil.release(msg);      ←
    }                                     通过调用 ReferenceCountUtil.release()
}                                         方法释放资源

```

扩展了
ChannelInboundHandlerAdapter

消费入站消息的简单方式 由于消费入站数据是一项常规任务，所以 Netty 提供了一个特殊的被称为 SimpleChannelInboundHandler 的 ChannelInboundHandler 实现。这个实现会在消息被 channelRead() 方法消费之后自动释放消息。

在出站方向这边，如果你处理了 write() 操作并丢弃了一个消息，那么你也应该负责释放它。代码清单 6-4 展示了一个丢弃所有的写入数据的实现。

代码清单 6-4 丢弃并释放出站消息

```

@Sharable
public class DiscardOutboundHandler
    extends ChannelOutboundHandlerAdapter {           ←
    @Override
    public void write(ChannelHandlerContext ctx,
                      Object msg, ChannelPromise promise) {
        ReferenceCountUtil.release(msg);      ←
        promise.setSuccess();                ←
    }
}

```

扩展了
ChannelOutboundHandlerAdapter

通过使用 ReferenceCountUtil.release(..)
方法释放资源

通知 ChannelPromise
数据已经被处理了

重要的是，不仅要释放资源，还要通知 ChannelPromise。否则可能会出现 ChannelFutureListener 收不到某个消息已经被处理了的通知的情况。

总之，如果一个消息被消费或者丢弃了，并且没有传递给 ChannelPipeline 中的下一个 ChannelOutboundHandler，那么用户就有责任调用 ReferenceCountUtil.release()。如果消息到达了实际的传输层，那么当它被写入时或者 Channel 关闭时，都将被自动释放。

6.2 ChannelPipeline 接口

如果你认为 ChannelPipeline 是一个拦截流经 Channel 的入站和出站事件的 ChannelHandler 实例链，那么就很容易看出这些 ChannelHandler 之间的交互是如何组成一个应用程序数据和事件处理逻辑的核心的。

每一个新创建的 Channel 都将会被分配一个新的 ChannelPipeline。这项关联是永久性的；Channel 既不能附加另外一个 ChannelPipeline，也不能分离其当前的。在 Netty 组件的生命周期中，这是一项固定的操作，不需要开发人员的任何干预。

根据事件的起源，事件将会被 ChannelInboundHandler 或者 ChannelOutboundHandler

处理。随后，通过调用 `ChannelHandlerContext` 实现，它将被转发给同一超类型的下一个 `ChannelHandler`。

ChannelHandlerContext

`ChannelHandlerContext` 使得 `ChannelHandler` 能够和它的 `ChannelPipeline` 以及其他 `ChannelHandler` 交互。`ChannelHandler` 可以通知其所属的 `ChannelPipeline` 中的下一个 `ChannelHandler`，甚至可以动态修改它所属的 `ChannelPipeline`^①。

`ChannelHandlerContext` 具有丰富的用于处理事件和执行 I/O 操作的 API。6.3 节将提供有关 `ChannelHandlerContext` 的更多内容。

图 6-3 展示了一个典型的同时具有入站和出站 `ChannelHandler` 的 `ChannelPipeline` 的布局，并且印证了我们之前的关于 `ChannelPipeline` 主要由一系列的 `ChannelHandler` 所组成的说法。`ChannelPipeline` 还提供了通过 `ChannelPipeline` 本身传播事件的方法。如果一个人站事件被触发，它将被从 `ChannelPipeline` 的头部开始一直被传播到 `ChannelPipeline` 的尾端。在图 6-3 中，一个出站 I/O 事件将从 `ChannelPipeline` 的最右边开始，然后向左传播。

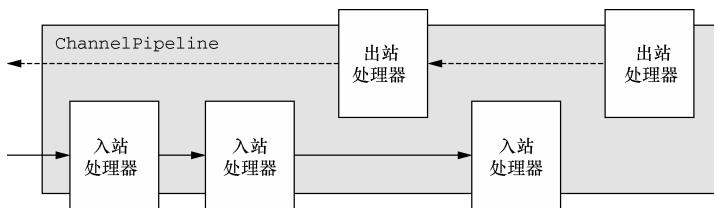


图 6-3 ChannelPipeline 和它的 ChannelHandler

ChannelPipeline 相对论

你可能会说，从事件途经 `ChannelPipeline` 的角度来看，`ChannelPipeline` 的头部和尾端取决于该事件是入站的还是出站的。然而 Netty 总是将 `ChannelPipeline` 的入站口（图 6-3 中的左侧）作为头部，而将出站口（该图的右侧）作为尾端。

当你完成了通过调用 `ChannelPipeline.add*`() 方法将入站处理器 (`ChannelInboundHandler`) 和出站处理器 (`ChannelOutboundHandler`) 混合添加到 `ChannelPipeline` 之后，每一个 `ChannelHandler` 从头部到尾端的顺序位置正如同我们方才所定义它们的一样。因此，如果你将图 6-3 中的处理器 (`ChannelHandler`) 从左到右进行编号，那么第一个被入站事件看到的 `ChannelHandler` 将是 1，而第一个被出站事件看到的 `ChannelHandler` 将是 5。

在 `ChannelPipeline` 传播事件时，它会测试 `ChannelPipeline` 中的下一个 `ChannelHandler` 的类型是否和事件的运动方向相匹配。如果不匹配，`ChannelPipeline` 将跳过该 `ChannelHandler` 并前进到下一个，直到它找到和该事件所期望的方向相匹配的为止。（当然，

^① 这里指修改 `ChannelPipeline` 中的 `ChannelHandler` 的编排。——译者注

ChannelHandler 也可以同时实现 ChannelInboundHandler 接口和 ChannelOutboundHandler 接口。)

6.2.1 修改 ChannelPipeline

ChannelHandler 可以通过添加、删除或者替换其他的 ChannelHandler 来实时地修改 ChannelPipeline 的布局。(它也可以将它自己从 ChannelPipeline 中移除。)这是 ChannelHandler 最重要的能力之一，所以我们将仔细地来看看它是如何做到的。表 6-6 列出了相关的方法。

表 6-6 ChannelHandler 的用于修改 ChannelPipeline 的方法

名 称	描 述
AddFirst addBefore addLast	将一个 ChannelHandler 添加到 ChannelPipeline 中
remove	将一个 ChannelHandler 从 ChannelPipeline 中移除
replace	将 ChannelPipeline 中的一个 ChannelHandler 替换为另一个 ChannelHandler

代码清单 6-5 展示了这些方法的使用。

代码清单 6-5 修改 ChannelPipeline

创建一个 FirstHandler 的实例

```
ChannelPipeline pipeline = ...;
-> FirstHandler firstHandler = new FirstHandler();
-> pipeline.addLast("handler1", firstHandler);
```

将该实例作为 "handler1" 添加到 ChannelPipeline 中

```
pipeline.addFirst("handler2", new SecondHandler());
-> pipeline.addLast("handler3", new ThirdHandler());
...
pipeline.remove("handler3");
pipeline.remove(firstHandler);
pipeline.replace("handler2", "handler4", new ForthHandler());
```

通过名称移除 "handler3"

通过引用移除

将一个 SecondHandler 的实例作为 "handler2" 添加到 ChannelPipeline 的第一个槽中。这意味着它将被放置在已有的 "handler1" 之前

将一个 ThirdHandler 的实例作为 "handler3" 替换为 FourthHandler。"handler4" 是唯一的，所以不需要它的名称

稍后，你将看到，重组 ChannelHandler 的这种能力使我们可以用它来轻松地实现极其灵活的逻辑。

ChannelHandler 的执行和阻塞

通常 ChannelPipeline 中的每一个 ChannelHandler 都是通过它的 EventLoop(I/O 线程)来处理传递给它的事件的。所以至关重要的是不要阻塞这个线程，因为这会对整体的 I/O 处理产生负面影响。

但有时可能需要与那些使用阻塞 API 的遗留代码进行交互。对于这种情况，ChannelPipeline 有一些接受一个 EventExecutorGroup 的 add() 方法。如果一个事件被传递给一个自定义的 EventExecutor-

Group，它将被包含在这个 EventExecutorGroup 中的某个 EventExecutor 所处理，从而被从该 Channel 本身的 EventLoop 中移除。对于这种用例，Netty 提供了一个叫 DefaultEventExecutorGroup 的默认实现。

除了这些操作，还有别的通过类型或者名称来访问 ChannelHandler 的方法。这些方法都列在了表 6-7 中。

表 6-7 ChannelPipeline 的用于访问 ChannelHandler 的操作

名 称	描 述
get	通过类型或者名称返回 ChannelHandler
context	返回和 ChannelHandler 绑定的 ChannelHandlerContext
names	返回 ChannelPipeline 中所有 ChannelHandler 的名称

6.2.2 触发事件

ChannelPipeline 的 API 公开了用于调用入站和出站操作的附加方法。表 6-8 列出了入站操作，用于通知 ChannelInboundHandler 在 ChannelPipeline 中所发生的事件。

表 6-8 ChannelPipeline 的入站操作

方 法 名 称	描 述
fireChannelRegistered	调用 ChannelPipeline 中下一个 ChannelInboundHandler 的 channelRegistered(ChannelHandlerContext) 方法
fireChannelUnregistered	调用 ChannelPipeline 中下一个 ChannelInboundHandler 的 channelUnregistered(ChannelHandlerContext) 方法
fireChannelActive	调用 ChannelPipeline 中下一个 ChannelInboundHandler 的 channelActive(ChannelHandlerContext) 方法
fireChannelInactive	调用 ChannelPipeline 中下一个 ChannelInboundHandler 的 channelInactive(ChannelHandlerContext) 方法
fireExceptionCaught	调用 ChannelPipeline 中下一个 ChannelInboundHandler 的 exceptionCaught(ChannelHandlerContext, Throwable) 方法
fireUserEventTriggered	调用 ChannelPipeline 中下一个 ChannelInboundHandler 的 userEventTriggered(ChannelHandlerContext, Object) 方法
fireChannelRead	调用 ChannelPipeline 中下一个 ChannelInboundHandler 的 channelRead(ChannelHandlerContext, Object msg) 方法
fireChannelReadComplete	调用 ChannelPipeline 中下一个 ChannelInboundHandler 的 channelReadComplete(ChannelHandlerContext) 方法
fireChannelWritabilityChanged	调用 ChannelPipeline 中下一个 ChannelInboundHandler 的 channelWritabilityChanged(ChannelHandlerContext) 方法

在出站这边，处理事件将会导致底层的套接字上发生一系列的动作。表 6-9 列出了 ChannelPipeline API 的出站操作。

表 6-9 ChannelPipeline 的出站操作

方法名称	描述
bind	将 Channel 绑定到一个本地地址，这将调用 ChannelPipeline 中的下一个 ChannelOutboundHandler 的 bind(ChannelHandlerContext, SocketAddress, ChannelPromise) 方法
connect	将 Channel 连接到一个远程地址，这将调用 ChannelPipeline 中的下一个 ChannelOutboundHandler 的 connect(ChannelHandlerContext, SocketAddress, ChannelPromise) 方法
disconnect	将 Channel 断开连接。这将调用 ChannelPipeline 中的下一个 ChannelOutboundHandler 的 disconnect(ChannelHandlerContext, ChannelPromise) 方法
close	将 Channel 关闭。这将调用 ChannelPipeline 中的下一个 ChannelOutboundHandler 的 close(ChannelHandlerContext, ChannelPromise) 方法
deregister	将 Channel 从它先前所分配的 EventExecutor (即 EventLoop) 中注销。这将调用 ChannelPipeline 中的下一个 ChannelOutboundHandler 的 deregister(ChannelHandlerContext, ChannelPromise) 方法
flush	冲刷 Channel 所有挂起的写入。这将调用 ChannelPipeline 中的下一个 ChannelOutboundHandler 的 flush(ChannelHandlerContext) 方法
write	将消息写入 Channel。这将调用 ChannelPipeline 中的下一个 ChannelOutboundHandler 的 write(ChannelHandlerContext, Object msg, ChannelPromise) 方法。注意：这并不会将消息写入底层的 Socket，而只会将它放入队列中。要将它写入 Socket，需要调用 flush() 或者 writeAndFlush() 方法
writeAndFlush	这是一个先调用 write() 方法再接着调用 flush() 方法的便利方法
read	请求从 Channel 中读取更多的数据。这将调用 ChannelPipeline 中的下一个 ChannelOutboundHandler 的 read(ChannelHandlerContext) 方法

总结一下：

- ChannelPipeline 保存了与 Channel 相关联的 ChannelHandler；
- ChannelPipeline 可以根据需要，通过添加或者删除 ChannelHandler 来动态地修改；
- ChannelPipeline 有着丰富的 API 用以被调用，以响应入站和出站事件。

6.3 ChannelHandlerContext 接口

ChannelHandlerContext 代表了 ChannelHandler 和 ChannelPipeline 之间的关联。每当有 ChannelHandler 添加到 ChannelPipeline 中时，都会创建 ChannelHandlerContext。ChannelHandlerContext 的主要功能是管理它所关联的 ChannelHandler 和在同一个 ChannelPipeline 中的其他 ChannelHandler 之间的交互。

ChannelHandlerContext 有很多的方法，其中一些方法也存在于 Channel 和 ChannelPipeline 本身上，但是有一点重要的不同。如果调用 Channel 或者 ChannelPipeline 上的这些方法，它们将沿着整个 ChannelPipeline 进行传播。而调用位于 ChannelHandlerContext 上的相同方法，则将从当前所关联的 ChannelHandler 开始，并且只会传播给位于该 ChannelPipeline 中的下一个能够处理该事件的 ChannelHandler。

表 6-10 对 ChannelHandlerContext API 进行了总结。

表 6-10 ChannelHandlerContext 的 API

方法名称	描述
alloc	返回和这个实例相关联的 Channel 所配置的 ByteBufAllocator
bind	绑定到给定的 SocketAddress，并返回 ChannelFuture
channel	返回绑定到这个实例的 Channel
close	关闭 Channel，并返回 ChannelFuture
connect	连接给定的 SocketAddress，并返回 ChannelFuture
deregister	从之前分配的 EventExecutor 注销，并返回 ChannelFuture
disconnect	从远程节点断开，并返回 ChannelFuture
executor	返回调度事件的 EventExecutor
fireChannelActive	触发对下一个 ChannelInboundHandler 上的 channelActive() 方法（已连接）的调用
fireChannelInactive	触发对下一个 ChannelInboundHandler 上的 channelInactive() 方法（已关闭）的调用
fireChannelRead	触发对下一个 ChannelInboundHandler 上的 channelRead() 方法（已接收的消息）的调用
fireChannelReadComplete	触发对下一个 ChannelInboundHandler 上的 channelReadComplete() 方法的调用
fireChannelRegistered	触发对下一个 ChannelInboundHandler 上的 fireChannelRegistered() 方法的调用
fireChannelUnregistered	触发对下一个 ChannelInboundHandler 上的 fireChannelUnregistered() 方法的调用
fireChannelWritabilityChanged	触发对下一个 ChannelInboundHandler 上的 fireChannelWritabilityChanged() 方法的调用
fireExceptionCaught	触发对下一个 ChannelInboundHandler 上的 fireExceptionCaught(Throwable) 方法的调用
fireUserEventTriggered	触发对下一个 ChannelInboundHandler 上的 fireUserEventTriggered(Object evt) 方法的调用
handler	返回绑定到这个实例的 ChannelHandler
isRemoved	如果所关联的 ChannelHandler 已经被从 ChannelPipeline 中移除则返回 true
name	返回这个实例的唯一名称
pipeline	返回这个实例所关联的 ChannelPipeline
read	将数据从 Channel 读取到第一个入站缓冲区；如果读取成功则触发 ^① 一个 channelRead 事件，并（在最后一个消息被读取完成后）通知 ChannelInboundHandler 的 channelReadComplete(ChannelHandlerContext) 方法

① 通过配合 ChannelConfig.setAutoRead(boolean autoRead) 方法，可以实现反应式系统的特性之一回压（back-pressure）。——译者注

续表

方法名称	描述
write	通过这个实例写入消息并经过 ChannelPipeline
writeAndFlush	通过这个实例写入并冲刷消息并经过 ChannelPipeline

当使用 ChannelHandlerContext 的 API 的时候, 请牢记以下两点:

- ChannelHandlerContext 和 ChannelHandler 之间的关联(绑定)是永远不会改变的, 所以缓存对它的引用是安全的;
- 如同我们在本节开头所解释的一样, 相对于其他类的同名方法, ChannelHandlerContext 的方法将产生更短的事件流, 应该尽可能地利用这个特性来获得最大的性能。

6.3.1 使用 ChannelHandlerContext

在这一节中我们将讨论 ChannelHandlerContext 的用法, 以及存在于 ChannelHandlerContext、Channel 和 ChannelPipeline 上的方法的行为。图 6-4 展示了它们之间的关系。

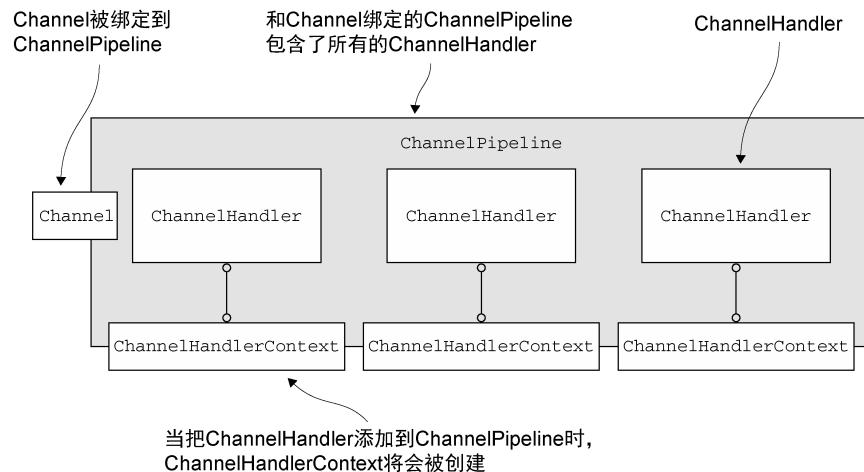


图 6-4 Channel、ChannelPipeline、ChannelHandler 以及 ChannelHandlerContext 之间的关系

在代码清单 6-6 中, 将通过 ChannelHandlerContext 获取到 Channel 的引用。调用 Channel 上的 write() 方法将会导致写入事件从尾端到头部地流经 ChannelPipeline。

代码清单 6-6 从 ChannelHandlerContext 访问 Channel

```
ChannelHandlerContext ctx = ...;
Channel channel = ctx.channel();
```

获取到与 ChannelHandlerContext
相关联的 Channel 的引用

```
channel.write(Unpooled.copiedBuffer("Netty in Action",
    CharsetUtil.UTF_8));
```

通过 Channel 写入
缓冲区

代码清单 6-7 展示了一个类似的例子，但是这一次是写入 ChannelPipeline。我们再次看到，(到 ChannelPipeline 的) 引用是通过 ChannelHandlerContext 获取的。

代码清单 6-7 通过 ChannelHandlerContext 访问 ChannelPipeline

```
ChannelHandlerContext ctx = ...;
ChannelPipeline pipeline = ctx.pipeline();
pipeline.write(Unpooled.copiedBuffer("Netty in Action",
    CharsetUtil.UTF_8));
```

获取到与 ChannelHandlerContext
相关联的 ChannelPipeline 的引用

通过 ChannelPipeline
写入缓冲区

如同在图 6-5 中所能够看到的一样，代码清单 6-6 和代码清单 6-7 中的事件流是一样的。重要的是要注意到，虽然被调用的 Channel 或 ChannelPipeline 上的 write() 方法将一直传播事件通过整个 ChannelPipeline，但是在 ChannelHandler 的级别上，事件从一个 ChannelHandler 到下一个 ChannelHandler 的移动是由 ChannelHandlerContext 上的调用完成的。

- ② 通过使用与之相关联的 ChannelHandlerContext，
ChannelHandler 将事件传递给了 ChannelPipeline
中的下一个 ChannelHandler
- ③ 通过使用与之相关联的 ChannelHandlerContext，
ChannelHandler 将事件传递给了 ChannelPipeline
中的下一个 ChannelHandler

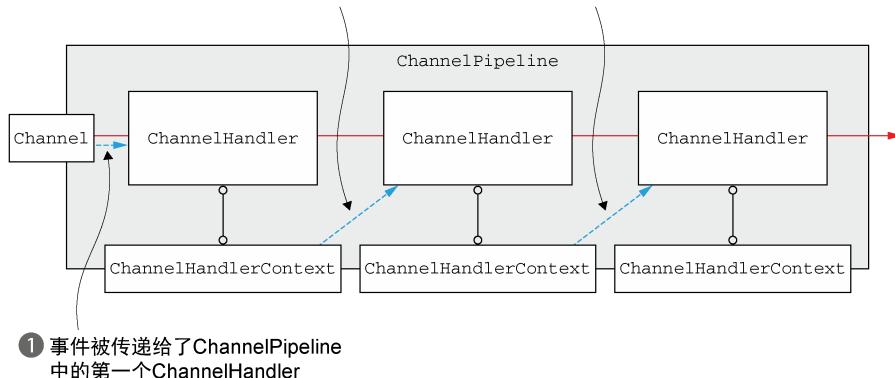


图 6-5 通过 Channel 或者 ChannelPipeline 进行的事件传播

为什么想要从 ChannelPipeline 中的某个特定点开始传播事件呢？

- 为了减少将事件传经对它不感兴趣的 ChannelHandler 所带来的开销。
- 为了避免将事件传经那些可能会对它感兴趣的 ChannelHandler。

要想调用从某个特定的 ChannelHandler 开始的处理过程，必须获取到在 (ChannelPipeline) 该 ChannelHandler 之前的 ChannelHandler 所关联的 ChannelHandlerContext。这个 ChannelHandlerContext 将调用和它所关联的 ChannelHandler 之后的 ChannelHandler。

代码清单 6-8 和图 6-6 说明了这种用法。

代码清单 6-8 调用 ChannelHandlerContext 的 write() 方法

```
ChannelHandlerContext ctx = ...;
ctx.write(Unpooled.copiedBuffer("Netty in Action", CharsetUtil.UTF_8));
```

↑
获取到 ChannelHandlerContext
的引用

write()方法将把缓冲区数据发送
到下一个 ChannelHandler

如图 6-6 所示，消息将从下一个 ChannelHandler 开始流经 ChannelPipeline，绕过了所有前面的 ChannelHandler。

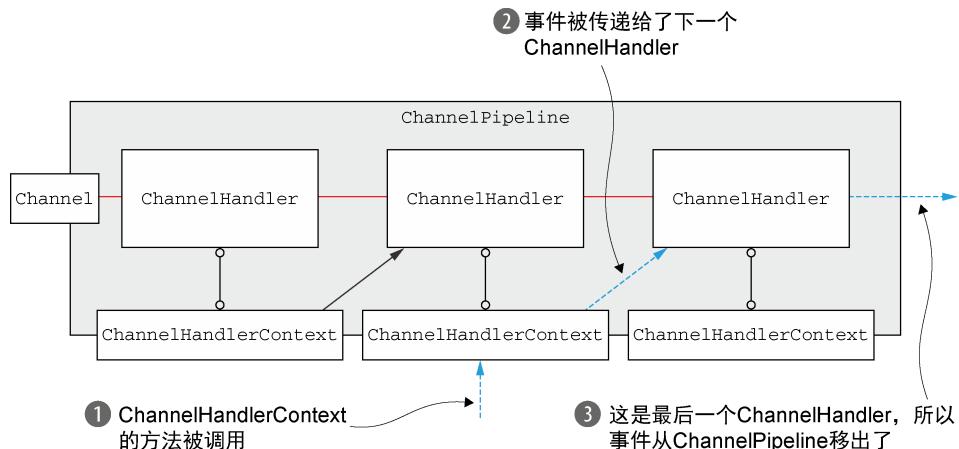


图 6-6 通过 ChannelHandlerContext 触发的操作的事件流

我们刚才所描述的用例是常见的，对于调用特定的 **ChannelHandler** 实例上的操作尤其有用。

6.3.2 ChannelHandler 和 ChannelHandlerContext 的高级用法

正如我们在代码清单 6-6 中所看到的，你可以通过调用 **ChannelHandlerContext** 上的 `pipeline()` 方法来获得被封闭的 **ChannelPipeline** 的引用。这使得运行时得以操作 **ChannelPipeline** 的 **ChannelHandler**，我们可以利用这一点来实现一些复杂的设计。例如，你可以通过将 **ChannelHandler** 添加到 **ChannelPipeline** 中来实现动态的协议切换。

另一种高级的用法是缓存到 **ChannelHandlerContext** 的引用以供稍后使用，这可能会发生在任何的 **ChannelHandler** 方法之外，甚至来自于不同的线程。代码清单 6-9 展示了用这种模式来触发事件。

代码清单 6-9 缓存到 ChannelHandlerContext 的引用

```

public class WriteHandler extends ChannelHandlerAdapter {
    private ChannelHandlerContext ctx;
    @Override
    public void handlerAdded(ChannelHandlerContext ctx) {
        this.ctx = ctx; ← 存储到 ChannelHandlerContext 的引用以供稍后使用
    }
    public void send(String msg) {
        ctx.writeAndFlush(msg); ← 使用之前存储的到 ChannelHandlerContext 的引用来发送消息
    }
}

```

因为一个 ChannelHandler 可以从属于多个 ChannelPipeline，所以它也可以绑定到多个 ChannelHandlerContext 实例。对于这种用法指在多个 ChannelPipeline 中共享同一个 ChannelHandler，对应的 ChannelHandler 必须要使用 @Sharable 注解标注；否则，试图将它添加到多个 ChannelPipeline 时将会触发异常。显而易见，为了安全地被用于多个并发的 Channel（即连接），这样的 ChannelHandler 必须是线程安全的。

代码清单 6-10 展示了这种模式的一个正确实现。

代码清单 6-10 可共享的 ChannelHandler

```

@Sharable
public class SharableHandler extends ChannelInboundHandlerAdapter { ← 使用注解
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) { ← @Sharable 标注
        System.out.println("Channel read message: " + msg);
        ctx.fireChannelRead(msg); ← 记录方法调用，并转发给下一个 ChannelHandler
    }
}

```

前面的 ChannelHandler 实现符合所有的将其加入到多个 ChannelPipeline 的需求，即它使用了注解 @Sharable 标注，并且也不持有任何的状态。相反，代码清单 6-11 中的实现将会导致问题。

代码清单 6-11 @Sharable 的错误用法

```

@Sharable
public class UnshareableHandler extends ChannelInboundHandlerAdapter { ← 使用注解
    private int count;
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) {
        count++;
        System.out.println("channelRead(...) called the " ← 将 count 字段的值加 1
            + count + " time");
        ctx.fireChannelRead(msg); ← 记录方法调用，并转发给下一个 ChannelHandler
    }
}

```

这段代码的问题在于它拥有状态^①，即用于跟踪方法调用次数的实例变量count。将这个类的一个实例添加到ChannelPipeline将极有可能在它被多个并发的Channel访问时导致问题。（当然，这个简单的问题可以通过使channelRead()方法变为同步方法来修正。）

总之，只应该在确定了你的ChannelHandler是线程安全的时才使用@Sharable注解。

为何要共享同一个ChannelHandler 在多个ChannelPipeline中安装同一个ChannelHandler的一个常见的原因是用于收集跨越多个Channel的统计信息。

我们对于ChannelHandlerContext和它与其他的框架组件之间的关系的讨论到此就结束了。接下来我们将看看异常处理。

6.4 异常处理

异常处理是任何真实应用程序的重要组成部分，它也可以通过多种方式来实现。因此，Netty提供了几种方式用于处理入站或者出站处理过程中所抛出的异常。这一节将帮助你了解如何设计最适合你需要的方式。

6.4.1 处理入站异常

如果在处理入站事件的过程中有异常被抛出，那么它将从它在ChannelInboundHandler里被触发的那一点开始流经ChannelPipeline。要想处理这种类型的人站异常，你需要在你的ChannelInboundHandler实现中重写下面的方法。

```
public void exceptionCaught(
    ChannelHandlerContext ctx, Throwable cause) throws Exception
```

代码清单 6-12 展示了一个简单的示例，其关闭了Channel并打印了异常的栈跟踪信息。

代码清单 6-12 基本的入站异常处理

```
public class InboundExceptionHandler extends ChannelInboundHandlerAdapter {
    @Override
    public void exceptionCaught(ChannelHandlerContext ctx,
        Throwable cause) {
        cause.printStackTrace();
        ctx.close();
    }
}
```

因为异常将会继续按照入站方向流动（就像所有的入站事件一样），所以实现了前面所示逻辑的ChannelInboundHandler通常位于ChannelPipeline的最后。这确保了所有的入站异常都总是会被处理，无论它们可能会发生在ChannelPipeline中的什么位置。

^① 主要的问题在于，对于其所持有的状态的修改并不是线程安全的，比如也可以通过使用AtomicInteger来规避这个问题。——译者注

你应该如何响应异常，可能很大程度上取决于你的应用程序。你可能想要关闭Channel（和连接），也可能尝试进行恢复。如果你不实现任何处理入站异常的逻辑（或者没有消费该异常），那么Netty将会记录该异常没有被处理的事实^①。

总结一下：

- ChannelHandler.exceptionCaught() 的默认实现是简单地将当前异常转发给 ChannelPipeline 中的下一个 ChannelHandler；
- 如果异常到达了 ChannelPipeline 的尾端，它将会被记录为未被处理；
- 要想定义自定义的处理逻辑，你需要重写 exceptionCaught() 方法。然后你需要决定是否需要将该异常传播出去。

6.4.2 处理出站异常

用于处理出站操作中的正常完成以及异常的选项，都基于以下的通知机制。

- 每个出站操作都将返回一个 ChannelFuture。注册到 ChannelFuture 的 ChannelFutureListener 将在操作完成时被通知该操作是成功了还是出错了。
- 几乎所有的 ChannelOutboundHandler 上的方法都会传入一个 ChannelPromise 的实例。作为 ChannelFuture 的子类，ChannelPromise 也可以被分配用于异步通知的监听器。但是，ChannelPromise 还具有提供立即通知的可写方法：

```
ChannelPromise setSuccess();
ChannelPromise setFailure(Throwable cause);
```

添加 ChannelFutureListener 只需要调用 ChannelFuture 实例上的 addListener(ChannelFutureListener) 方法，并且有两种不同的方式可以做到这一点。其中最常用的方式是，调用出站操作（如 write() 方法）所返回的 ChannelFuture 上的 addListener() 方法。

代码清单 6-13 使用了这种方式来添加 ChannelFutureListener，它将打印栈跟踪信息并且随后关闭 Channel。

代码清单 6-13 添加 ChannelFutureListener 到 ChannelFuture

```
ChannelFuture future = channel.write(someMessage);
future.addListener(new ChannelFutureListener() {
    @Override
    public void operationComplete(ChannelFuture f) {
        if (!f.isSuccess()) {
            f.cause().printStackTrace();
            f.channel().close();
        }
    }
});
```

^① 即 Netty 将会通过 Warning 级别的日志记录该异常到达了 ChannelPipeline 的尾端，但没有被处理，并尝试释放该异常。——译者注

第二种方式是将 `ChannelFutureListener` 添加到即将作为参数传递给 `ChannelOutboundHandler` 的方法的 `ChannelPromise`。代码清单 6-14 中所展示的代码和代码清单 6-13 中所展示的具有相同的效果。

代码清单 6-14 添加 `ChannelFutureListener` 到 `ChannelPromise`

```
public class OutboundExceptionHandler extends ChannelOutboundHandlerAdapter {
    @Override
    public void write(ChannelHandlerContext ctx, Object msg,
                      ChannelPromise promise) {
        promise.addListener(new ChannelFutureListener() {
            @Override
            public void operationComplete(ChannelFuture f) {
                if (!f.isSuccess()) {
                    f.cause().printStackTrace();
                    f.channel().close();
                }
            }
        });
    }
}
```

ChannelPromise 的可写方法

通过调用 `ChannelPromise` 上的 `setSuccess()` 和 `setFailure()` 方法，可以使一个操作的状态在 `ChannelHandler` 的方法返回给其调用者时便即刻被感知到。

为何选择一种方式而不是另一种呢？对于细致的异常处理，你可能会发现，在调用出站操作时添加 `ChannelFutureListener` 更合适，如代码清单 6-13 所示。而对于一般的异常处理，你可能会发现，代码清单 6-14 所示的自定义的 `ChannelOutboundHandler` 实现的方式更加的简单。

如果你的 `ChannelOutboundHandler` 本身抛出了异常会发生什么呢？在这种情况下，Netty 本身会通知任何已经注册到对应 `ChannelPromise` 的监听器。

6.5 小结

在本章中我们仔细地研究了 Netty 的数据处理组件——`ChannelHandler`。我们讨论了 `ChannelHandler` 是如何链接在一起，以及它们是如何作为 `ChannelInboundHandler` 和 `ChannelOutboundHandler` 与 `ChannelPipeline` 进行交互的。

下一章将介绍 Netty 的 `EventLoop` 和并发模型，这对于理解 Netty 是如何实现异步的、事件驱动的网络编程模型来说至关重要。