原地址：https://github.com/netty/netty/wiki/User-guide-for-4.x
作者：高俊（116445039@qq.com）
转载请注明出处！谢谢！

问题：
目前我们经常使用能满足我们目标的应用程序或框架，实现和其他应用的通讯。比如，我们经常使用HttpClient这个框架从一个Web服务器中检索数据，或者通过web service进行远程过程调用（RPC）。但是，一个通过的协议或者一个具体的实现有时不可扩展。比如，我们不使用http服务器进行传输大文件、email消息、财务系统或多玩家的准实时信息。需要一个高度优化的协议去实现一个专一的特殊目标。比如，你可能想要实现一个http服务器，用于优化基于ajax的聊天应用、多媒体流或大文件传输。你甚至想要去设计并实现一个能量身定做满足你需求的新协议。另外一个不可避免的问题时，有时你必须使用一个遗留的协议与老系统进行交互。在这种情况下，关系系统的问题是如何快速的实现这个协议并且不牺牲系统的稳定性和性能。

解决方案：
Netty努力提供一种异步的消息驱动的网络应用框架，作为一种快速开发可维护的高性能高可扩展性服务器和客户端的工具。
另外，Netty是一个可以快速、简单开发CS结构网络应用的NIO客户端、服务器端框架。它能极大简化、流水线化基于TCP、UDP Socket服务器的网络编程。
"快而简"并不会影响应用程序的可维护性和性能。Netty吸取了大量的二级制的、文本的已有协议的设计经验，比如FTP、SMTP、HTTP，并进行了仔细的设计。因此，Netty能够以一种没有折中的方案实现系统开发的简易性、高性能、稳定性及灵活性。
有些用户已经知道一些声称具有同样优势的应用框架，你可能会问Netty和他们有什么区别。答案是Netty是建立在哲学之上的。Netty设计了一系列的API和快速的实现，让你最舒适的开发体验。它不是一些真实触摸的东西，但是你使用Netty时会意识到一种令你的生活更轻松的哲学原理，就像你阅读这份说明指南一样。

起步：
为了让你能够让你快速起步，这一章通过简单的样例遍历Netty的核心架构。当这一章节结束的时候，你将能利用Netty基础上创建一个客户端和一个服务器。
如果你喜欢至上而下的学习方式，可以跳到章节2学习完Netty的整体架构之后再回到本章节。

开始之前：
为了能够运行本章节介绍的样例，至少需要满足两个要求：最新版本的Netty和JDK1.6以上版本。可以在Netty的下载页面找到最新版的Netty。为了下载合适的JDK，请参考你喜好的JDK提供者的网站。
在你阅读的过程中，可能会碰见更多的关于本章介绍的Class的问题。当你想知道这些Class的更多信息，请随时查看API手册。为了方便你查看，文档中所有的Class名称都有对应API手册的链接。另外，当你发现有错误的信息、错误的语法和排版，或者你有改进这个文档的建议，请一定在Netty交流区让我们知道。

编写一个Discard服务器：
世界上最简单的协议不是"Hello,World!"，而是DISCARD。他是一个丢失所有接收到的数据而没有回应的协议。
为了实现DISCARD协议，你只需要做一件事就是忽略所有接收到的数据。让我们直接以实现handler开始，它用于处理由Netty产生的I/O事件。

```
package io.netty.example.discard;

import io.netty.buffer.ByteBuf;

import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.ChannelInboundHandlerAdapter;

/**
 * Handles a server-side channel.
 */
public class DiscardServerHandler extends ChannelInboundHandlerAdapter { // (1)

    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) { // (2)
        // Discard the received data silently.
        ((ByteBuf) msg).release(); // (3)
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) { // (4)
        // Close the connection when an exception is raised.
        cause.printStackTrace();
        ctx.close();
    }
}
```

1.DiscardServerHandler继承ChannelInboundHandlerAdapter，其是ChannelInboundHandler的一种实现。ChannelInboundHandler提供了大量的可用于重载的事件处理函数。目前，你只需要继承ChannelInboundHandlerAdapter就足够了，不用自己去实现接口中的函数。
2.这里我们重载了channelRead这个事件处理函数。当有新消息从客户端接收到时，这个函数会在被调用并且传递已接收到的消息。在这个样例中，接收消息的类型是ByteBuff。
3.为了实现DISCARD协议，处理函数必须忽略接收到的消息。ByteBuff是一种按引用计数的对象，必须通过调用release()才能释放。请注意handler的职责是释放任何传入的按引用计数的对象。一般情况下，channelRead函数的实现如下：

```
@Override
public void channelRead(ChannelHandlerContext ctx, Object msg) {
    try {
        // Do something with msg
    } finally {
        ReferenceCountUtil.release(msg);
    }
}
```

4.当Netty在处理IO产生异常或一个事件处理函数处理事件时抛出异常时，exceptionCaught()事件处理函数会被调用并传递一个Throwable对象。大部分情况下，捕获的异常应该被记录日志并关闭相关的通道（channel），除非这个方法中能够通过依赖的异常情况来进行不同的实现。比如，你可能想要在关闭连接之前，发送一个附带异常状态码的响应消息。

截止目前，我们已经实现了DISCARD服务器的一半。接下来要做的是编写main方法用于启用包含DiscardServerHandler的服务器。

```
package io.netty.example.discard;

import io.netty.bootstrap.ServerBootstrap;

import io.netty.channel.ChannelFuture;
import io.netty.channel.ChannelInitializer;
import io.netty.channel.ChannelOption;
import io.netty.channel.EventLoopGroup;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.SocketChannel;
import io.netty.channel.socket.nio.NioServerSocketChannel;

/**
 * Discards any incoming data.
 */
public class DiscardServer {

    private int port;

    public DiscardServer(int port) {
        this.port = port;
    }

    public void run() throws Exception {
        EventLoopGroup bossGroup = new NioEventLoopGroup(); // (1)
        EventLoopGroup workerGroup = new NioEventLoopGroup();
        try {
            ServerBootstrap b = new ServerBootstrap(); // (2)
            b.group(bossGroup, workerGroup)
             .channel(NioServerSocketChannel.class) // (3)
             .childHandler(new ChannelInitializer<SocketChannel>() { // (4)
                 @Override
                 public void initChannel(SocketChannel ch) throws Exception {
                     ch.pipeline().addLast(new DiscardServerHandler());
                 }
             })
             .option(ChannelOption.SO_BACKLOG, 128)          // (5)
             .childOption(ChannelOption.SO_KEEPALIVE, true); // (6)

            // Bind and start to accept incoming connections.
            ChannelFuture f = b.bind(port).sync(); // (7)

            // Wait until the server socket is closed.
            // In this example, this does not happen, but you can do that to gracefully
            // shut down your server.
            f.channel().closeFuture().sync();
        } finally {
            workerGroup.shutdownGracefully();
            bossGroup.shutdownGracefully();
        }
    }

    public static void main(String[] args) throws Exception {
        int port;
        if (args.length > 0) {
            port = Integer.parseInt(args[0]);
        } else {
            port = 8080;
        }
        new DiscardServer(port).run();
    }
}
```

1.NioEventLoopGroup是一个处理IO操作的多线程事件环路。Netty提供了大量的EventLoopGroup实现，用于处理不同类型的传输。在这个样例中我们正在实现服务器端应用程序，因此我们要使用两个NioEventLoopGroup。第一个常被叫做"boss"，用于接收接入的连接。第二个常被叫做"work"，处理一经boss接受并注册到work的连接。需要使用多少线程以及如何将它们映射到已经创建的通道上，则依赖于EventLoopGroup中可配置的构造函数。
2.ServerBootstrap是一个用于启动服务器的帮助类。你可以使用通道直接启动服务器。但是，请知晓这是一个无趣的过程，大部分请下你不需要这样处理。
3.这里我们明确的使用NioServerSocketChannel类用于实例化一个通道，用于接受接入的连接。
4.这里明确的处理函数有一个新接受的通道来评估。ChannelInitializer是一个特殊的处理函数，它用来帮助用户配置一个新的通道。你可能想通过增加一些像DiscardServerHandler的处理函数来配置一个新创建通道的ChannelPipeline，用于实现你的网络应用程序。当应用程序变的复杂的过程中，你最终可能要在顶层类中增加增加更多的管道（pipeline）的处理函数和匿名类。
5.你也可以设置通道实现中指定的参数。我们在实现TCP/IP的服务器，所以我们允许去设置一些socket的选项，比如tcpNoDelay、keepAlive。请参考ChannelOption的Api文档和ChannelConfig实现说明，对已支持的ChannelOptions有一个整体的了解。
6.你是否意识到option()和childOption()？option()是用于接受接入的连接的NioSocketChannel。childOption()是用于被ServerChannel接受的通道，在这里指NioServerSocketChannel。
7.我们已经准备完毕。剩下要做的是绑定端口和启动服务器。这里我们绑定端口在8080，现在你可以在不同的绑定地址上调用bind()方法任意次。
祝贺你！你已经完成了建立在Netty上你的第一个服务器。

查看接收到的数据：
我们已经写了我们的第一个服务器，接下来我们需要测试它是否真的能工作。最简单的测试方式是使用Telnet命令。比如，你可以在命令行下输入telnet localhost 8080并且输入一些内容。
但是，这样我们就能认为服务器能正常工作了吗？我们实际上没有确认因为它是一个Discard服务器。你不会获得任何响应。为了证实服务器能够正常工作，让我们修改服务器，输出接收到的内容。
我们已经清楚当接收到数据时会调用channelRead()方法。让我们在DiscardServerHandler的channelRead()方法中增加一些代码。

```
@Override
public void channelRead(ChannelHandlerContext ctx, Object msg) {
    ByteBuf in = (ByteBuf) msg;
    try {
        while (in.isReadable()) { // (1)
            System.out.print((char) in.readByte());
            System.out.flush();
        }
    } finally {
        ReferenceCountUtil.release(msg); // (2)
    }
}
```

1.这个低效的循环，实际上可以简化为`System.out.println(in.toString(io.netty.util.CharsetUtil.US_ASCII))`
2.或者你可以在.release()中输出。
如果你再次运行telnet命令，你将看到服务器输出接收的信息。
discard server完整的源代码在`io.netty.example.discard`这个发布包中。

编写一个回声（echo）服务器：
截止目前，我们还没有在处理数据时做出响应。但是，一个服务器经常是认为应该对一个请求做出回应的。让我们学习如何通过实现ECHO协议而对一个客户端做出一个响应信息，在这里认为接收到是数据被原样返回。
与之前的Discard Server不同的是我们要在之前实现接口的逻辑——将接收的数据输出到控制台，修改为将数据发送返回。因此，我们将channelRead()方法修改为：

```
@Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) {
        ctx.write(msg); // (1)
        ctx.flush(); // (2)
    }
```

1.一个ChannelHandlerContext对象提供了大量操作方法，可以允许你触发大量的IO事件和操作。这里我们调用write()方法逐个的写出消息。请注意我们没有想discard server中那样释放接收到的消息。这是因为Netty会释放那些写出的通道。
2.使用cxt.write()方法不会将消息写出到通道。它会在内部缓存，当调用cxt.flush()时会刷出到通道中。另外，你可以调用简单调用ctx.writeAndFlush()。
当你再次运行telnet命令行，你将会收到你发送的任何内容。
echo server的完整代码在发布文件的`io.netty.example.echo`包里。

编写一个时间服务器：
这里我们要实现的Time协议，不同之前样例的是它不接收任何请求，它会发送一个保护32位的整数的消息，当消息发送之后即断开连接。在这个样例中，你会学习如何构造并发送一个消息，以及如何在发送完成之后关闭连接。
由于我们呢会忽悠任何接收到的消息，只在建立连接之后尽快返回一个消息，所以我们不能再使用channelRead()，而应该重载channelActive()。以下是方法的实现：

```
package io.netty.example.time;

public class TimeServerHandler extends ChannelInboundHandlerAdapter {

    @Override
    public void channelActive(final ChannelHandlerContext ctx) { // (1)
        final ByteBuf time = ctx.alloc().buffer(4); // (2)
        time.writeInt((int) (System.currentTimeMillis() / 1000L + 2208988800L));

        final ChannelFuture f = ctx.writeAndFlush(time); // (3)
        f.addListener(new ChannelFutureListener() {
            @Override
            public void operationComplete(ChannelFuture future) {
                assert f == future;
                ctx.close();
            }
        }); // (4)
    }

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

1.和之前的介绍一样，channelActive()方法将会在连接创建完成并准备发送数据后调用。在这个方法中，我们写出一个代表当前时间的32位整数。
2.为了发送消息，我们必须创建一个新的ByteBuff用于保存消息。由于我们要写出一个32位的整数，所以我们需要一个容量至少为4字节的ByteBuff。通过ChannelHandlerContext.alloc()方法获取当前的ByteBuffAllocator，再用它创建ByteBuff。
3.像平常一样，我们写出构造好的消息。
但是等等，在什么地方做弹出（flip）呢？我们不是经常要在使用NIO时，在发送消息之前要先调用`java.nio.ByteBuffer.flip()`吗？ByteBuff没有这个方法，因为它有两个指针，一个用于读操作，另一个用于写操作。写的下标会在写入时增加，而读的下标不会发生变化。读下标和写下标分别表示消息的开始和结束。
与之不同的，NIO buffer没有提供明确的方法指出消息是从哪开始和结束的，除非调用flip方法。如果没有数据或错误的数据被发送，而你忘记调用flip方法，你可能就碰到麻烦了。这种错误不会出现在Netty中，因为我们为不同的操作类型提供了不同的指针。你将会感觉生活更加简易，当你熟悉了它——一种不需要flip out的生活方式。
另一个需要说明的指针是ChannelContextHandler.write(或writeAndFlush())方法返回的ChannelFuture。一个ChannelFuture表示一个还没有发生的IO操作。它表示，任何的操作可能还没有执行，因为在Netty中所有的操作都是异步的。比如，下面的代码可能在发送一个消息之前闭关了连接：

```
Channel ch = ...;
ch.writeAndFlush(message);
ch.close();
```

所以，你需要在ChannelFuture完成之后再调用close方法，ChannelFuture是由write方法返回的，并且它会在操作完成之后通知和它相关的监听器（Listener）。请注意，close方法可能也不会立即关闭连接，而是返回一个ChannelFuture对象。
4.当一个请求完成之后，我们将如何获取通知呢？很简单，只需要为返回的ChannelFuture添加一个ChannelFutureListener。这里我们创建了一个匿名的ChannelFutureListner用于在操作完成之后关闭连接。
另一种方式，你可以使用预定义的listener简化代码：

```
f.addListener(ChannelFutureListener.CLOSE);
```

为测试时间服务器是否按预计的方式工作，你可以使用Unix中的rdate命令：

```
$ rdate -o <port> -p <host>
```

port是你在main方法中设置的端口号，host常是localhost。

编写一个时间客户端：
与DISCARD和ECHO服务器不同，我们需要一个用于Time协议的客户端，因为人们无法将一个32位的二进制数据转换为一个日历时间。在这一节，我们将学习如何确保这个服务器工作正常并且学习如何使用Netty编写一个客户端。
在服务器和客户端之间最大的也是唯一的不同是，实现不同的Bootstrap和Channel。请查看如下的代码：

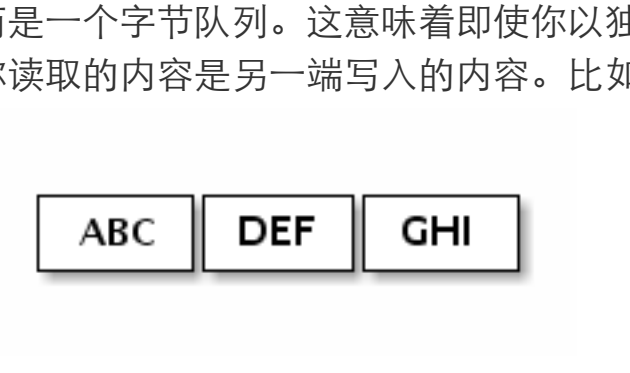
```
package io.netty.example.time;

public class TimeClient {
    public static void main(String[] args) throws Exception {
        String host = args[0];
        int port = Integer.parseInt(args[1]);
        EventLoopGroup workerGroup = new NioEventLoopGroup();

        try {
            Bootstrap b = new Bootstrap(); // (1)
            b.group(workerGroup); // (2)
            b.channel(NioSocketChannel.class); // (3)
            b.option(ChannelOption.SO_KEEPALIVE, true); // (4)
            b.handler(new ChannelInitializer<SocketChannel>() {
                @Override
                public void initChannel(SocketChannel ch) throws Exception {
                    ch.pipeline().addLast(new TimeClientHandler());
                }
            });

            // Start the client.
            ChannelFuture f = b.connect(host, port).sync(); // (5)

            // Wait until the connection is closed.
            f.channel().closeFuture().sync();
        } finally {
            workerGroup.shutdownGracefully();
        }
    }
}
```

1.Bootstrap和ServerBootstrap很相似，不同在于它是用于非服务器的通道，比如客户端或无连接的通道。
2.如果只设置了一个EventLoopGroup，它将被同时用于Boss组和Work组。当然，在客户端不需要使用Boss work。
3.取代NioServerSocketChannel，使用NioSocketChannel来创建一个客户端的通道。
4.注意，我们没有像ServerBootstrap使用childOption方法，因为客户端的SocketChannel没有父通道。
5.我们应该调用connect方法，而不是bind方法。
如你所见，它真的和服务器端的代码没有什么区别。ChannelHandler的实现呢？它应该接收来至服务器端的32位的整数，将它转换为可阅读的格式，输出转换后的时间并关闭连接：

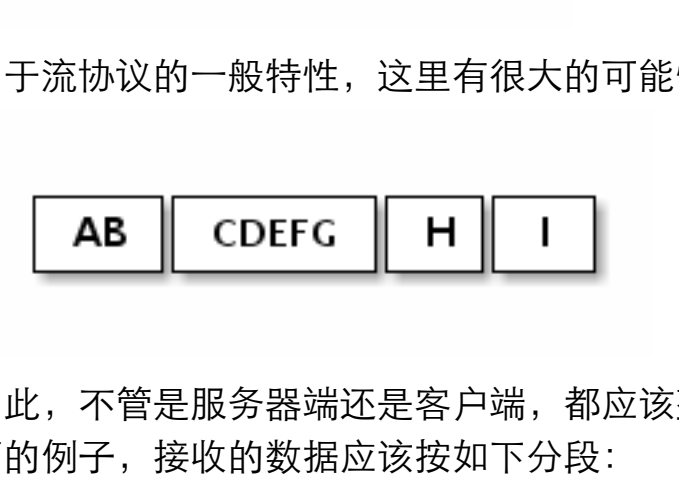
```
package io.netty.example.time;

import java.util.Date;

public class TimeClientHandler extends ChannelInboundHandlerAdapter {
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) {
        ByteBuf m = (ByteBuf) msg; // (1)
        try {
            long currentTimeMillis = (m.readUnsignedInt() - 2208988800L) * 1000L;
            System.out.println(new Date(currentTimeMillis));
            ctx.close();
        } finally {
            m.release();
        }
    }

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

1.在TCP/IP中，Netty从管道中读取数据到ByteBuff中。
它看上去很简单，并且和服务器端的例子没有什么区别。但是，这个Handler有时会因为`IndexOutOfBoundsException`而拒绝工作。下一节中我们将讨论为什么会出现这种问题。

处理基于流的传输：
一个关于Socket缓冲的小警告
在如TCP/IP基于流的传输，接收到的数据会被缓存在一个Socket接收缓冲中。不幸的是，这个基于流传输的缓冲不是一个包队列，而是一个字节队列。这意味着即使你以独立的包发送两个数据，操作系统不会把它们当做两个消息而是一系列的字节。因此，不保证你读取的内容是另一端写入的内容。比如，让我们假设操作系统的TCP/IP栈接收了三个包：



由于流协议的一般特性，这里有很大的可能性，在你的应用程序中你读取的片段是如下的形式：



因此，不管是服务器端还是客户端，都应该整理接收到的数据为一个或多个有意义的片段，这些片段能容易的被应用程序理解。像上面的例子，接收的数据应该按如下分段：

第一种解决方案：
现在让我们回到上面的时间（TIME）客户端样例。我们应该面临相同的问题。一个32位的整数是一个很小量的数据，并且它常常不会被分段。但是，问题是它可以被分段，并且当流量增加时分段的可能性也会增加。
最简单的方案是在创建一个内部的递增的缓冲，等待直到所有的4个字节都被接收到这个内部的缓冲。以下是修复了这个问题，修改之后的TimeClientHandler实现：

```
package io.netty.example.time;

import java.util.Date;

public class TimeClientHandler extends ChannelInboundHandlerAdapter {
    private ByteBuf buf;

    @Override
    public void handlerAdded(ChannelHandlerContext ctx) {
        buf = ctx.alloc().buffer(4); // (1)
    }

    @Override
    public void handlerRemoved(ChannelHandlerContext ctx) {
        buf.release(); // (1)
        buf = null;
    }

    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) {
        ByteBuf m = (ByteBuf) msg;
        buf.writeBytes(m); // (2)
        m.release();

        if (buf.readableBytes() >= 4) { // (3)
            long currentTimeMillis = (buf.readUnsignedInt() - 2208988800L) * 1000L;
            System.out.println(new Date(currentTimeMillis));
            ctx.close();
        }
    }

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

1.一个ChannelHandler有两个生命周期监听方法：handlerAdded和handleRemove。你可以设置任意的初始化（销毁）任务和它没有阻塞很久一样。
2.首先，所有接收的数据应该被累积到buf中。
3.然后，handler必须校验buf是否接收足够的数据，比如样例中的4个字节，并且用于实际应用程序的处理。如果没有接收足够的数据，Netty将会再次调用channelRead方法，接收更多的数据，最终累计到4个字节。

第二个解决方案：
虽然第一个方法解决了TIME客户端中的问题，但是修改之后的handler看上去不够简洁。想象一下，如果有一个更复杂的协议，它是由多个不固定长度的域组成的。你的ChannelInbounderHandler实现很快会变得无法维护。
就如你可能意识到的，你可以为一个ChannelPipeline设置多个ChannelHandler，所以你可以将一个整体的ChannelHandler分成多个模块以降低应用程序的复杂度。比如，你可以把TimeClientHandler分为两个handler：
- TimeDecoder用于处理分段功能，以及
- TimeClientHandler简单版本的初始化

幸运的是，Netty提供了一个扩展类帮助你实现第一部分：

```
package io.netty.example.time;

public class TimeDecoder extends ByteToMessageDecoder { // (1)
    @Override
    protected void decode(ChannelHandlerContext ctx, ByteBuf in, List<Object> out) { // (2)
        if (in.readableBytes() < 4) {
            return; // (3)
        }

        out.add(in.readBytes(4)); // (4)
    }
}
```

1.ByteToMessageDecoder是一个ChannelInboundHandler的实现，用于简化分段功能的处理。
2.ByteToMessageDecoder会在接收数据时调用decode方法，它使用一个内部的可维护的累计缓冲。
3.decode会不会在没有累计缓冲足够的数据前增加到out中。ByteToMessageDecoder会当接收到更多的数据时调用decode方法。
4.如果decode增加一个对象到out中，则表示解码器（decoder）解码成功。ByteToMessage就会忽略累计缓冲中剩余的读取到的部分。请记住，你不需求解码多个消息。ByteToMessageDecoder将会持续的调用decode方法，知道没有内容增加到out中。
现在我们有另一个handler加入到ChannelPipeline，我们需要修改TimeClient中的ChannelInitializer实现：

```
b.handler(new ChannelInitializer<SocketChannel>() {
    @Override
    public void initChannel(SocketChannel ch) throws Exception {
        ch.pipeline().addLast(new TimeDecoder(), new TimeClientHandler());
    }
});
```

如果你是一个爱折腾的人，你可能想要尝试RelayingDecoder，进一步简化解码器。然而你将需要请教API参考手册获取更多信息。

```
public class TimeDecoder extends ReplayingDecoder<Void> {
    @Override
    protected void decode(
            ChannelHandlerContext ctx, ByteBuf in, List<Object> out) {
        out.add(in.readBytes(4));
    }
}
```

另外，Netty提供了解包（out-of-the-box）的解码器，它能使你很简单的实现大部分协议，并且帮助你避免以一个大而难维护的handler实现而结束。请参考以下包中的样例：
- `io.netty.example.factorial` for a binary protocol, and
- `io.netty.example.telnet` for a text line-based protocol.

替代ByteBuf使用POJO对象进行通话：
截止目前我们复习的样例都是以ByteBuf作为一个协议消息的主要数据结构。在这一章节，我们将会改进TIME客户端，使用POJO类替代ByteBuf。
在ChannelHandler中使用POJO类的优势是明显的。通过将handler中从分离ByteBuf中提取信息的代码分离出来，你的handler将会更易于维护和复用。在TIME客户端和服务器的样例中，我们只读取一个32位的整数，它不是直接使用ByteBuf主要方式。但是，你将会发现有必要将它分离出来以实现一个词（word）协议。
首要，让我们定义个一个叫UnixTime的新类型。

```
package io.netty.example.time;

import java.util.Date;

public class UnixTime {

    private final long value;

    public UnixTime() {
        this(System.currentTimeMillis() / 1000L + 2208988800L);
    }

    public UnixTime(long value) {
        this.value = value;
    }

    public long value() {
        return value;
    }

    @Override
    public String toString() {
        return new Date((value() - 2208988800L) * 1000L).toString();
    }
}
```

我们可以修订TimeDecoder创建一个UnixTime来替代ByteBuf。

```
@Override
protected void decode(ChannelHandlerContext ctx, ByteBuf in, List<Object> out) {
    if (in.readableBytes() < 4) {
        return;
    }

    out.add(new UnixTime(in.readUnsignedInt()));
}
```

使用更新之后的解码器，TimeClientHandler不再需要使用ByteBuf：

```
@Override
public void channelRead(ChannelHandlerContext ctx, Object msg) {
    UnixTime m = (UnixTime) msg;
    System.out.println(m);
    ctx.close();
}
```

是否能更简单和优雅一些呢？相同的技术可以应用在服务器端。让我们第一次更新TimeServerHandler：

```
@Override
public void channelActive(ChannelHandlerContext ctx) {
    ChannelFuture f = ctx.writeAndFlush(new UnixTime());
    f.addListener(ChannelFutureListener.CLOSE);
}
```

现在，是只剩下的一点点是一个编码器，它用用于实现ChannelOutboundHandler将UnixTime转换为ByteBuf。它比编写一个解码器更简单，因为不需要在编码时去处理包的分段和组装。

```
package io.netty.example.time;

public class TimeEncoder extends ChannelOutboundHandlerAdapter {
    @Override
    public void write(ChannelHandlerContext ctx, Object msg, ChannelPromise promise) {
        UnixTime m = (UnixTime) msg;
        ByteBuf encoded = ctx.alloc().buffer(4);
        encoded.writeInt((int)m.value());
        ctx.write(encoded, promise); // (1)
    }
}
```

1.在这一行有一个重要的事情。
首先，我们传递了原始的ChannelPromise，因此当转码数据写入到通道时，Netty能标识它成功还是失败。
其次，我们没有调用ctx.flush()。有一个void flush（`ChannelHandlerContext ctx`）方法，为了重载flush()方法。
为了更简化代码，你可以使用MessageToByteEncoder：

```
public class TimeEncoder extends MessageToByteEncoder<UnixTime> {
    @Override
    protected void encode(ChannelHandlerContext ctx, UnixTime msg, ByteBuf out) {
        out.writeInt((int)msg.value());
    }
}
```

剩余的最后一步任务是在服务器端，在TimeServerHandler之前将一个TimeEncoder增加到ChannelPipeline，这里遗留作为一个不重要的练习。

关闭你的应用程序：
关闭一个Netty应用程序，通常像关闭所有通过`shutdownGracefully`方法创建的`EventLoopGroups`一样简单。它返回一个`Future`对象，它会在`EventLoopGroup`终止及其所有的通道关闭之后通知你。

总结：
在这一章节，我们通过在Netty之上编写一个完全可以工作的网络应用程序，来快速学习了Netty。在接下来的章节中还有更多关于Netty的详细信息。我们也鼓励你复习`io.netty.example`包中的Netty样例。
请也同时留意交流区，期待你的意见和建议来帮助我们持续改进Netty和相关文档。