# User guide for 4.x

## Preface

### The Problem

Nowadays we use general purpose applications or libraries to communicate with each other. For example, we often use an HTTP client library to retrieve information from a web server and to invoke a remote procedure call via web services. However, a general purpose protocol or its implementation sometimes does not scale very well. It is like we don't use a general purpose HTTP server to exchange huge files, e-mail messages, and near-realtime messages such as financial information and multiplayer game data. What's required is a highly optimized protocol implementation which is dedicated to a special purpose. For example, you might want to implement an HTTP server which is optimized for AJAX-based chat application, media streaming, or large file transfer. You could even want to design and implement a whole new protocol which is precisely tailored to your need. Another inevitable case is when you have to deal with a legacy proprietary protocol to ensure the interoperability with an old system. What matters in this case is how quickly we can implement that protocol while not sacrificing the stability and performance of the resulting application.

## The Solution

The Netty project is an effort to provide an asynchronous event-driven network application framework and tooling for the rapid development of maintainable high-performance • high-scalability protocol servers and clients.

In other words, Netty is an NIO client server framework which enables quick and easy development of network applications such as protocol servers and clients. It greatly simplifies and streamlines network programming such as TCP and UDP socket server development.

'Quick and easy' does not mean that a resulting application will suffer from a maintainability or a performance issue. Netty has been designed carefully with the experiences earned from the implementation of a lot of protocols such as FTP, SMTP, HTTP, and various binary and text-based legacy protocols. As a result, Netty has succeeded to find a way to achieve ease of development, performance, stability, and flexibility without a compromise.

Some users might already have found other network application framework that claims to have the same advantage, and you might want to ask what makes Netty so different from them. The answer is the philosophy it is built on. Netty is designed to give you the most comfortable experience both in terms of the API and the implementation from the day one. It is not something tangible but you will realize that this philosophy will make your life much easier as you read this guide and play with Netty.

# Getting Started

This chapter tours around the core constructs of Netty with simple examples to let you get started quickly. You will be able to write a client and a server on top of Netty right away when you are at the end of this chapter.

If you prefer top-down approach in learning something, you might want to start from Chapter 2, Architectural Overview and get back here.

## Before Getting Started

The minimum requirements to run the examples which are introduced in this chapter are only two; the latest version of Netty and JDK 1.6 or above. The latest version of Netty is available in the project download page. To download the right version of JDK, please refer to your preferred JDK vendor's web site.

As you read, you might have more questions about the classes introduced in this chapter. Please refer to the API reference whenever you want to know more about them. All class names in this document are linked to the online API reference for your convenience. Also, please don't hesitate to contact the Netty project community and let us know if there's any incorrect information, errors in grammar and typo, and if you have a good idea to improve the documentation.

## Writing a Discard Server

The most simplistic protocol in the world is not 'Hello, World!' but DISCARD. It's a protocol which discards any received data without any response.

To implement the DISCARD protocol, the only thing you need to do is to ignore all received data. Let us start straight from the handler implementation, which handles I/O events generated by Netty.

```
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 extends ChannelInboundHandlerAdapter, which is an implementation of ChannelInboundHandler. ChannelInboundHandler provides various event handler methods that you can override. For now, it is just enough to extend ChannelInboundHandlerAdapter rather than to implement the handler interface by yourself.
2. We override the channelRead() event handler method here. This method is called with the received message, whenever new data is received from a client. In this example, the type of the received message is ByteBuf.

3. To implement the DISCARD protocol, the handler has to ignore the received message. ByteBuf is a reference-counted object which has to be released explicitly via the release() method. Please keep in mind that it is the handler's responsibility to release any reference-counted object passed to the handler. Usually, channelRead() handler method is implemented like the following:

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

4. The exceptionCaught() event handler method is called with a Throwable when an exception was raised by Netty due to an I/O error or by a handler implementation due to the exception thrown while processing events. In most cases, the caught exception should be logged and its associated channel should be closed here, although the implementation of this method can be different depending on what you want to do to deal with an exceptional situation. For example, you might want to send a response message with an error code before closing the connection.

So far so good. We have implemented the first half of the DISCARD server. What's left now is to write the main() method which starts the server with the DiscardServerHandler.

```
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 is a multithreaded event loop that handles I/O operation. Netty provides various EventLoopGroup implementations for different kind of transports. We are implementing a server-side application in this example, and therefore two NioEventLoopGroup will be used. The first one, often called 'boss', accepts an incoming connection. The second one, often called 'worker', handles the traffic of the accepted connection once the boss accepts the connection and registers the accepted connection to the worker. How many Threads are used and how they are mapped to the created Channels depends on the EventLoopGroup implementation and may be even configurable via a constructor.
2. ServerBootstrap is a helper class that sets up a server. You can set up the server using a Channel directly. However, please note that this is a tedious process, and you do not need to do that in most cases.
3. Here, we specify to use the NioServerSocketChannel class which is used to instantiate a new Channel to accept incoming connections.
4. The handler specified here will always be evaluated by a newly accepted Channel. The ChannelInitializer is a special handler that is purposed to help a user configure a new Channel. It is most likely that you want to configure the ChannelPipeline of the new Channel by adding some handlers such as DiscardServerHandler to implement your network application. As the application gets complicated, it is likely that you will add more handlers to the pipeline and extract this

anonymous class into a top level class eventually.

5. You can also set the parameters which are specific to the `Channel` implementation. We are writing a TCP/IP server, so we are allowed to set the socket options such as `tcpNoDelay` and `keepAlive`. Please refer to the apidocs of `ChannelOption` and the specific `ChannelConfig` implementations to get an overview about the supported `ChannelOption`s.
6. Did you notice `option()` and `childOption()`? `option()` is for the `NioServerSocketChannel` that accepts incoming connections. `childOption()` is for the `Channel`s accepted by the parent `ServerChannel`, which is `NioServerSocketChannel` in this case.
7. We are ready to go now. What's left is to bind to the port and to start the server. Here, we bind to the port 8080 of all NICs (network interface cards) in the machine. You can now call the `bind()` method as many times as you want (with different bind addresses.)

Congratulations! You've just finished your first server on top of Netty.

## Looking into the Received Data

Now that we have written our first server, we need to test if it really works. The easiest way to test it is to use the telnet command. For example, you could enter `telnet localhost 8080` in the command line and type something.

However, can we say that the server is working fine? We cannot really know that because it is a discard server. You will not get any response at all. To prove it is really working, let us modify the server to print what it has received.

We already know that `channelRead()` method is invoked whenever data is received. Let us put some code into the `channelRead()` method of the `DiscardServerHandler`:

```
@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. This inefficient loop can actually be simplified to: `System.out.println(in.toString(io.netty.util.CharsetUtil.US_ASCII))`
2. Alternatively, you could do `in.release()` here.

If you run the telnet command again, you will see the server prints what has received.

The full source code of the discard server is located in the `io.netty.example.discard` package of the distribution.

## Writing an Echo Server

So far, we have been consuming data without responding at all. A server, however, is usually supposed to respond to a request. Let us learn how to write a response

message to a client by implementing the ECHO protocol, where any received data is sent back.

The only difference from the discard server we have implemented in the previous sections is that it sends the received data back instead of printing the received data out to the console. Therefore, it is enough again to modify the channelRead() method:

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

1. A ChannelHandlerContext object provides various operations that enable you to trigger various I/O events and operations. Here, we invoke write(Object) to write the received message in verbatim. Please note that we did not release the received message unlike we did in the DISCARD example. It is because Netty releases it for you when it is written out to the wire.
2. ctx.write(Object) does not make the message written out to the wire. It is buffered internally, and then flushed out to the wire by ctx.flush(). Alternatively, you could call ctx.writeAndFlush(msg) for brevity.

If you run the telnet command again, you will see the server sends back whatever you have sent to it.

The full source code of the echo server is located in the io.netty.example.echo package of the distribution.

## Writing a Time Server

The protocol to implement in this section is the TIME protocol. It is different from the previous examples in that it sends a message, which contains a 32-bit integer, without receiving any requests and loses the connection once the message is sent. In this example, you will learn how to construct and send a message, and to close the connection on completion.

Because we are going to ignore any received data but to send a message as soon as a connection is established, we cannot use the channelRead() method this time. Instead, we should override the channelActive() method. The following is the implementation:

```
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. As explained, the `channelActive()` method will be invoked when a connection is established and ready to generate traffic. Let's write a 32-bit integer that represents the current time in this method.
2. To send a new message, we need to allocate a new buffer which will contain the message. We are going to write a 32-bit integer, and therefore we need a `ByteBuf` whose capacity is at least 4 bytes. Get the current `ByteBufAllocator` via `ChannelHandlerContext.alloc()` and allocate a new buffer.

3. As usual, we write the constructed message.

   But wait, where's the flip? Didn't we used to call `java.nio.ByteBuffer.flip()` before sending a message in NIO? `ByteBuf` does not have such a method because it has two pointers; one for read operations and the other for write operations. The writer index increases when you write something to a `ByteBuf` while the reader index does not change. The reader index and the writer index represents where the message starts and ends respectively.

   In contrast, NIO buffer does not provide a clean way to figure out where the message content starts and ends without calling the flip method. You will be in trouble when you forget to flip the buffer because nothing or incorrect data will be sent. Such an error does not happen in Netty because we have different pointer for different operation types. You will find it makes your life much easier as you get used to it -- a life without flipping out!

   Another point to note is that the `ChannelHandlerContext.write()` (and `writeAndFlush()`) method returns a `ChannelFuture`. A `ChannelFuture` represents an I/O operation which has not yet occurred. It means, any requested operation might not have been performed yet because all operations are asynchronous in Netty. For example, the following code might close the connection even before a message is sent:

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

   Therefore, you need to call the `close()` method after the `ChannelFuture` is complete, which was returned by the `write()` method, and it notifies its listeners when the write operation has been done. Please note that, `close()` also might not close the connection immediately, and it returns a `ChannelFuture`.

4. How do we get notified when a write request is finished then? This is as simple as adding a `ChannelFutureListener` to the returned `ChannelFuture`. Here, we created a new anonymous `ChannelFutureListener` which closes the `Channel` when the operation is done.

   Alternatively, you could simplify the code using a pre-defined listener:

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

To test if our time server works as expected, you can use the UNIX `rdate` command:

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

where `<port>` is the port number you specified in the `main()` method and `<host>` is usually `localhost`.

## Writing a Time Client

Unlike `DISCARD` and `ECHO` servers, we need a client for the `TIME` protocol because a human cannot translate a 32-bit binary data into a date on a calendar. In this section, we discuss how to make sure the server works correctly and learn how to write a client with Netty.

The biggest and only difference between a server and a client in Netty is that different `Bootstrap` and `Channel` implementations are used. Please take a look at the following code:

```
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` is similar to `ServerBootstrap` except that it's for non-server channels such as a client-side or connectionless channel.
2. If you specify only one `EventLoopGroup`, it will be used both as a boss group and as a worker group. The boss worker is not used for the client side though.
3. Instead of `NioServerSocketChannel`, `NioSocketChannel` is being used to create a client-side `Channel`.
4. Note that we do not use `childOption()` here unlike we did with `ServerBootstrap` because the client-side `SocketChannel` does not have a parent.

5. We should call the `connect()` method instead of the `bind()` method.

As you can see, it is not really different from the the server-side code. What about the ChannelHandler implementation? It should receive a 32-bit integer from the server, translate it into a human readable format, print the translated time, and close the connection:

```
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. In TCP/IP, Netty reads the data sent from a peer into a ~~`ByteBuf`~~.

It looks very simple and does not look any different from the server side example. However, this handler sometimes will refuse to work raising an `IndexOutOfBoundsException`. We discuss why this happens in the next section.

## Dealing with a Stream-based Transport

### One Small Caveat of Socket Buffer

In a stream-based transport such as TCP/IP, received data is stored into a socket receive buffer. Unfortunately, the buffer of a stream-based transport is not a queue of packets but a queue of bytes. It means, even if you sent two messages as two independent packets, an operating system will not treat them as two messages but as just a bunch of bytes. Therefore, there is no guarantee that what you read is exactly what your remote peer wrote. For example, let us assume that the TCP/IP stack of an operating system has received three packets:


Because of this general property of a stream-based protocol, there's high chance of reading them in the following fragmented form in your application:


Therefore, a receiving part, regardless it is server-side or client-side, should defrag the received data into one or more meaningful frames that could be easily understood by the application logic. In case of the example above, the received data should be framed like the following:


## The First Solution

Now let us get back to the TIME client example. We have the same problem here. A 32-bit integer is a very small amount of data, and it is not likely to be fragmented often. However, the problem is that it can be fragmented, and the possibility of fragmentation will increase as the traffic increases.

The simplistic solution is to create an internal cumulative buffer and wait until all 4 bytes are received into the internal buffer. The following is the modified `TimeClientHandler` implementation that fixes the problem:

```
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. A ChannelHandler has two life cycle listener methods: handlerAdded() and handlerRemoved(). You can perform an arbitrary (de)initialization task as long as it does not block for a long time.
2. First, all received data should be cumulated into buf.
3. And then, the handler must check if buf has enough data, 4 bytes in this example, and proceed to the actual business logic. Otherwise, Netty will call the channelRead() method again when more data arrives, and eventually all 4 bytes will be cumulated.

## The Second Solution

Although the first solution has resolved the problem with the TIME client, the modified handler does not look that clean. Imagine a more complicated protocol which is composed of multiple fields such as a variable length field. Your ChannelInboundHandler implementation will become unmaintainable very quickly.

As you may have noticed, you can add more than one ChannelHandler to a ChannelPipeline, and therefore, you can split one monolithic ChannelHandler into multiple modular ones to reduce the complexity of your application. For example, you could split TimeClientHandler into two handlers:

- TimeDecoder which deals with the fragmentation issue, and
- the initial simple version of TimeClientHandler.

Fortunately, Netty provides an extensible class which helps you write the first one out of the box:

```
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 is an implementation of ChannelInboundHandler which makes it easy to deal with the fragmentation issue.
2. ByteToMessageDecoder calls the decode() method with an internally maintained cumulative buffer whenever new data is received.
3. decode() can decide to add nothing to out where there is not enough data in the cumulative buffer. ByteToMessageDecoder will call decode() again when there is more data received.
4. If decode() adds an object to out, it means the decoder decoded a message successfully. ByteToMessageDecoder will discard the read part of the cumulative

buffer. Please remember that you don't need to decode multiple messages. ByteToMessageDecoder will keep calling the decode() method until it adds nothing to out.

Now that we have another handler to insert into the ChannelPipeline, we should modify the ChannelInitializer implementation in the TimeClient:

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

If you are an adventurous person, you might want to try the ReplayingDecoder which simplifies the decoder even more. You will need to consult the API reference for more information though.

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

Additionally, Netty provides out-of-the-box decoders which enables you to implement most protocols very easily and helps you avoid from ending up with a monolithic unmaintainable handler implementation. Please refer to the following packages for more detailed examples:

- io.netty.example.factorial for a binary protocol, and
- io.netty.example.telnet for a text line-based protocol.

## Speaking in POJO instead of ByteBuf

All the examples we have reviewed so far used a ByteBuf as a primary data structure of a protocol message. In this section, we will improve the TIME protocol client and server example to use a POJO instead of a ByteBuf.

The advantage of using a POJO in your ChannelHandlers is obvious; your handler becomes more maintainable and reusable by separating the code which extracts information from ByteBuf out from the handler. In the TIME client and server examples, we read only one 32-bit integer and it is not a major issue to use ByteBuf directly. However, you will find it is necessary to make the separation as you implement a real world protocol.

First, let us define a new type called UnixTime.

```
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();
    }
}
```

We can now revise the `TimeDecoder` to produce a `UnixTime` instead of a `ByteBuf`.

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

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

With the updated decoder, the `TimeClientHandler` does not use `ByteBuf` anymore:

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

Much simpler and elegant, right? The same technique can be applied on the server side. Let us update the `TimeServerHandler` first this time:

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

Now, the only missing piece is an encoder, which is an implementation of `ChannelOutboundHandler` that translates a `UnixTime` back into a `ByteBuf`. It's much simpler than writing a decoder because there's no need to deal with packet fragmentation and assembly when encoding a message.

```
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. There are quite a few important things in this single line.

   First, we pass the original ChannelPromise as-is so that Netty marks it as success or failure when the encoded data is actually written out to the wire.

   Second, we did not call ctx.flush(). There is a separate handler method void flush(ChannelHandlerContext ctx) which is purposed to override the flush() operation.

To simplify even further, you can make use of MessageToByteEncoder:

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

The last task left is to insert a TimeEncoder into the ChannelPipeline on the server side before the TimeServerHandler, and it is left as a trivial exercise.

## Shutting Down Your Application

Shutting down a Netty application is usually as simple as shutting down all EventLoopGroups you created via shutdownGracefully(). It returns a Future that notifies you when the EventLoopGroup has been terminated completely and all Channels that belong to the group have been closed.

## Summary

In this chapter, we had a quick tour of Netty with a demonstration on how to write a fully working network application on top of Netty.

There is more detailed information about Netty in the upcoming chapters. We also encourage you to review the Netty examples in the io.netty.example package.

Please also note that the community is always waiting for your questions and ideas to help you and keep improving Netty and its documentation based on your feedback.

Last retrieved on 10-Nov-2015

---