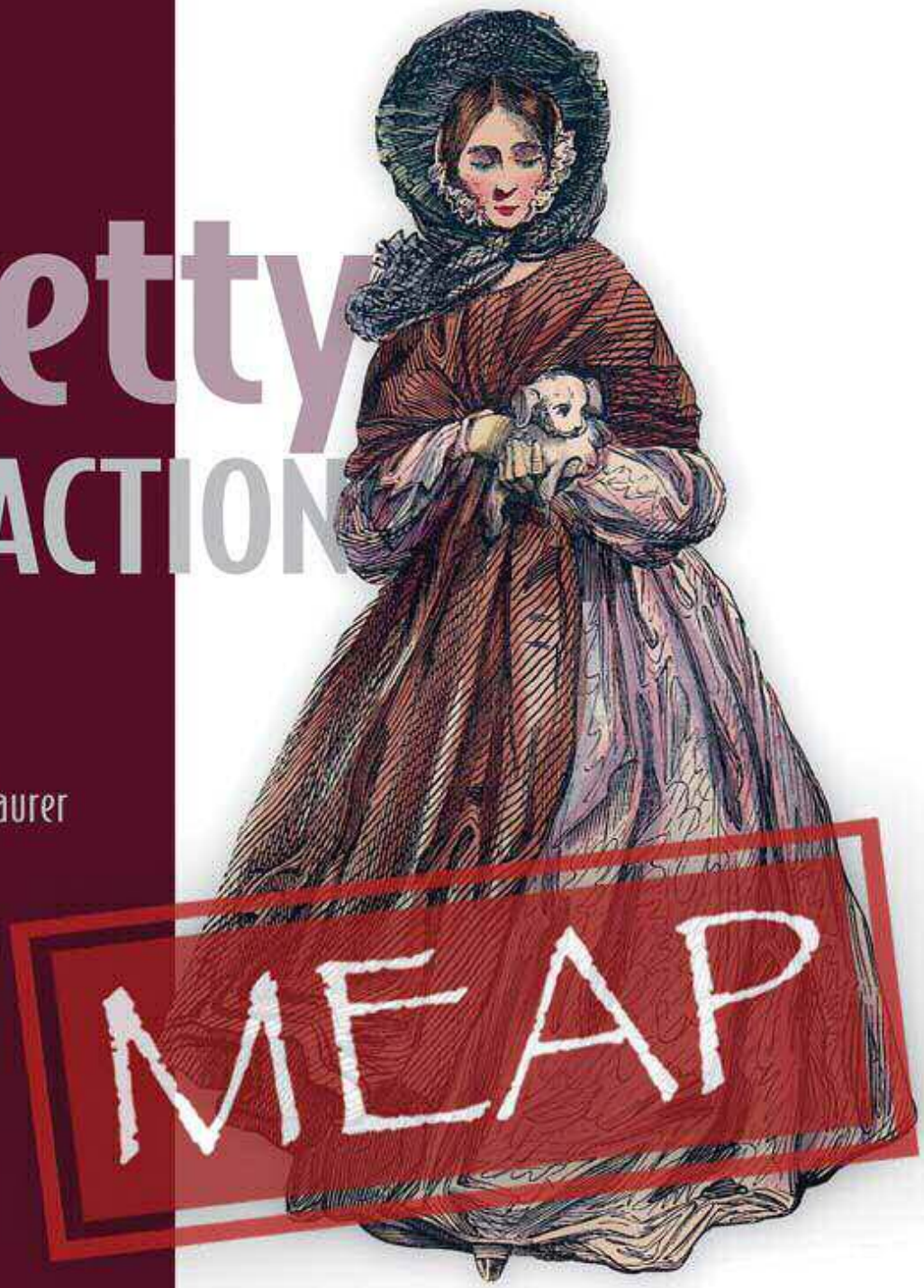
Netty
IN ACTION

Norman Maurer



MANNING

MEAP Edition
Manning Early Access Program
Netty in Action
Version 5

For more information on this and other Manning titles go to
www.manning.com

brief contents

1 Netty and Java NIO APIs

This chapter covers

- Netty architecture
- Why we need non-blocking IO (NIO)
- Blocking versus non-blocking IO
- Known problems with the JDK's NIO implementation and Netty's solutions

This chapter introduces Netty, but it's focus is Java's non-blocking IO (NIO) API. If you're new to networking on the JVM, this chapter is an ideal place to begin, but it's also a good refresher for the seasoned Java developer. If you're familiar with NIO and NIO.2, feel free to skip ahead to chapter 2, which dives into Netty after you get it running on your machine.

Netty is a NIO client-server framework, which enables quick and easy development of network applications, such as protocol servers and clients. Netty offers you a new way to develop your network applications, which makes it easy and scalable. It achieves this by abstracting away the complexity involved and by providing an easy-to-use API that decouples business-logic from the network-handling code. Because it's built for NIO, the entire Netty API is asynchronous.

Generally, network applications have scalability issues, whether they're based on Netty or other NIO APIs. A key component of Netty is its asynchronous nature, and this chapter discusses synchronous (blocking) and asynchronous (non-blocking) IO to illustrate why and how asynchronous code solves scalability problems.

For those new to networking, this chapter gives you an understanding of networking applications, in general, and how Netty helps implement them. It explains how to use fundamental Java networking APIs, discusses both its strengths and weaknesses, and shows how Netty addresses Java issues, such as the `Epoll` bug or memory leaks.

By the end of the chapter, you'll understand what Netty is and what it offers, and you'll gain enough understanding about Java's NIO and asynchronous processing to enable you to work through the other chapters of this book.

1.1 Why Netty?

In the words of David Wheeler, "all problems in computer science can be solved by another level of indirection."[1] As an NIO client-server framework, Netty offers one such level of indirection. Netty simplifies network programming of TCP or UDP servers but you can still access and use the low-level APIs because Netty provides high-level abstractions.

1.1.1 Not all networking frameworks are alike

With Netty, "quick and easy" doesn't mean that a resulting application suffers from maintainability or performance issues. The experiences earned from the implementation of protocols such as FTP, SMTP, HTTP, WebSocket, SPDY and various binary and text-based legacy protocols led Netty's founders to take great care in its design. As a result, Netty successfully delivers ease of development, performance, stability, and flexibility without compromise.

High-profile companies and open-source projects including RedHat, Twitter, Infinispan, and HornetQ, Vert.x, Finagle, Akka, Apache Cassandra, Elasticsearch, and others use and contribute to Netty. It's also fair to say that some of the features of Netty are a result of the needs of these projects. Over the years, Netty has become more widely known and is one of the most-used networking frameworks for the JVM, as evident by its use in several popular open- and closed-source projects. In fact, Netty was awarded the Duke's Choice Award[2] in 2011.

Also in 2011, Netty founder Trustin Lee left RedHat to join Twitter. At this point, the Netty project became independent of any company in an effort to simplify contributions to it. Both Red Hat and Twitter use Netty, so it should come as no surprise that these two companies are the biggest contributors to Netty at the time of writing. Adoption of Netty continues to increases, as does the number of individual contributors. The community of Netty users is active and the project remains vibrant.

1.1.2 Netty boasts a rich feature set

As you work through the chapters of this book, you'll learn about and use many of Netty's features. Figure 1.1 highlights some of the supported transports and protocols to give you a brief overview of the Netty architecture.

[1] Add citation
[2] Add citation

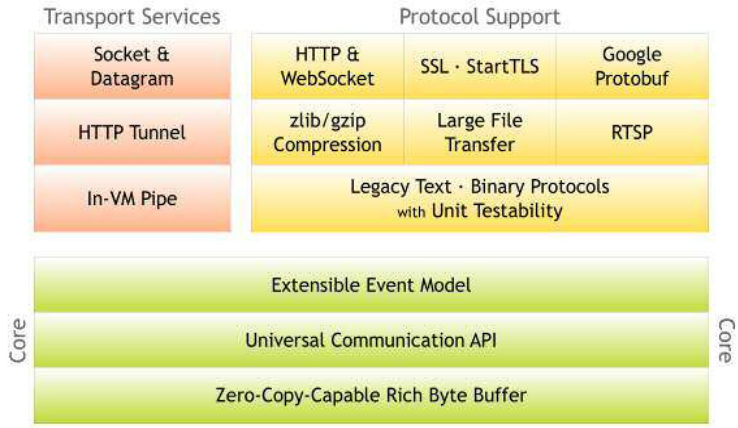


Figure 1.1 Overview of the model, transports, and protocols

In addition to its wide array of transports and protocols, Netty offers benefits in various development areas (see table 1.1).

Table 1.1 Netty gives developers a full set of tools.

Development Area	Netty Features
Design	▪ Unified API for various transport types—blocking and non-blocking socket
	▪ Flexible to use
	▪ Simple but powerful thread-model
	▪ True connectionless datagram socket support
	▪ Chaining of logics to make reuse easy
Ease of Use	▪ Well-documented Javadoc and plenty of examples provided
	▪ No additional dependencies except JDK 1.6 (or above). Some features are supported only in Java 1.7 and above. Other features may have other dependencies, but these are optional
Performance	▪ Better throughput; lower latency than core Java APIs
	▪ Less resource consumption because of pooling and reuse
	▪ Minimized unnecessary memory copy
Robustness	▪ No more `OutOfMemoryError` due to fast, slow, or overloaded connection.
	▪ No more unfair read/write ratio often found in a NIO application in high-speed

	networks
Security	▪ Complete SSL/TLS and StartTLS support
	▪ Runs in a restricted environment such as Applet or OSGI
Community	▪ Release early, release often
	▪ Active

In addition to the features listed, Netty also ships workarounds for known bugs/limitations in Java's NIO, so you don't need to hassle with them.

With an overview of Netty's features under your belt, it's time to take a closer look at asynchronous processing and the idea behind it. NIO and Netty make heavy use of asynchronous code, and without understanding the impact of the choices associated with this, it's hard to get the best out of it. In the next section, we'll take a look at why we need asynchronous APIs.

1.2 Asynchronous by design

The entire Netty API is asynchronous. Asynchronous processing isn't new; the idea has been around for a while. These days, however, IO is often a bottleneck, and asynchronous processing is becoming more important every day. But how does it work and what are the different patterns available?

Asynchronous processing encourages you to use your resources more efficiently by allowing you to start a task and get notified when it's done rather than waiting for it to complete. You're free to do something else while the task is running.

This section explains the two most common ways to work with or implement an asynchronous API and discusses the differences between the techniques.

1.2.1 Callbacks

Callbacks are a technique often used with asynchronous processing. A callback is passed to the method and executes after the method completes. You may recognize this pattern from JavaScript, in which callbacks are at the heart of the language. The following listing shows how to use this technique to fetch data.

Listing 1.1 Callback example

```
public interface Fetcher {
    void fetchData(FetchCallback callback);
}

public interface FetchCallback {
    void onData(Data data);
    void onError(Throwable cause);
}

public class Worker {
    public void doWork() {
```

```
        Fetcher fetcher = ...
        fetcher.fetchData(new FetchCallback() {
            @Override
            public void onData(Data data) {                              #1
                System.out.println("Data received: " + data);
            }

            @Override
            public void onError(Throwable cause) {                       #2
                System.err.println("An error accour: " + cause.getMessage());
            }
        });
        ...
    }
}
```

#1 Call if data is fetched without error
#2 Call if error is received during fetch

The `Fetcher.fetchData()` method takes one argument of type `FetchCallback`, which is called when either the data is fetched or an error occurs.

For each of these situations, it provides one method:

- `FetchCallback.onData()`—Called if the data was fetched without an error (#1)
- `FetchCallback.onError()`—Called if an error was received during the fetch operation (#2)

You can, therefore, move the execution of these methods from the "caller" thread to some other thread. There is no guarantee whenever one of the methods of the `FetchCallback` will be called.

One problem with the callback approach is that it can lead to spaghetti code when you chain many asynchronous method calls with different callbacks. Some people tend to think this approach results in hard-to-read code, but I think it's more a matter of taste and style. For example, Node.js, which is based on JavaScript, is increasingly popular. It makes heavy use of callbacks, yet many people find that it's easy to read and write applications with.

1.2.2 Futures

A second technique is the use of Futures. A Future is an abstraction, which represents a value that may become available at some point. A `Future` object either holds the result of a computation or, in the case of a failed computation, an exception.

Java ships with a `Future` interface in the java.util.concurrent package, which uses it by its Executor for asynchronous processing.

For example, as shown in the next listing, whenever you pass a `Runnable` object to the `ExecutorService.submit()` method, you get a `Future` back, which you can use to check if the execution completed.

Listing 1.2 Future example via `ExecutorService`

```
ExecutorService executor = Executors.newCachedThreadPool();
Runnable task1 = new Runnable() {
```

```
    @Override
    public void run() {
        doSomeHeavyWork();
    }
    ...
}
Callable<Interger> task2 = new Callable() {

    @Override
    public Integer call() {
        return doSomeHeavyWorkWithResul();
    }
    ...
}

Future<?> future1 = executor.submit(task1);
Future<Integer> future2 = executor.submit(task2);
while (!future1.isDone() || !future2.isDone()) {
    ...
    // do something else
    ...
}
 #A
 #B
```

You can also use this technique in your own API. For example, you could implement a Fetcher (as in listing 1.1) that uses a Future. As shown in the following listing.

Listing 1.3 Future usage for `Fetcher`

```
public interface Fetcher {
    Future<Data> fetchData();
}

public class Worker {
    public void doWork() {
        Fetcher fetcher = ...
        Future<Data> future = fetcher.fetchData();
        try {
            while(!fetcher.isDone()) {
                ...
                // do something else
            }
System.out.println("Data received: " + future.get());

        } catch (Throwable cause) {
                System.err.println("An error accour: " +
cause.getMessage());

        }
    }
}
 #A
 #B
```

Again, you check if the fetcher is done yet and if not, do some other work. Sometimes using futures can feel ugly because you need to check the state of the Future in intervals to see if it is completed yet, whereas with a callback you're notified directly after it's done.

After seeing both common techniques for asynchronous execution, you may wonder which is the best. There is no clear answer here. Netty, in fact, uses a mixture of the two to provide you with the best of both worlds.

The next section provides an introduction to writing networking applications on the JVM first using blocking, then using the NIO and NIO.2 APIs. These foundations are essential for getting the most out of subsequent chapters of this book. If you're familiar with the Java networking APIs, then perhaps only a quick scan of the next section is required to refresh your memory.

1.3 Blocking versus non-blocking IO on the JVM

The continued growth of the web has increased the need for networking applications that are capable of handling its scale. Efficiency has become important in meeting these demands. Fortunately, Java comes with the tools needed to create efficient, scalable networking applications. Although early versions of Java included networking support, it was Java 1.4 that introduced the NIO API and paved the way for us to write more efficient networking applications.

The new API (NIO.2), introduced in Java 7, was designed to allow us to write asynchronous networking code but tries to provide a more high-level API than does its predecessor.

To do networking-related tasks in Java, you can take one of two approaches:

- use IO , also known as blocking IO
- use NIO, also known as new/non-blocking IO

New or non-blocking?

The *N* in NIO is typically thought to mean "non-blocking" rather than "new."NIO has been around for so long now that nobody calls it "new" IO anymore. Most people refer to it as "non-blocking" IO—

Figure 1.2 shows how the blocking IO uses one dedicated thread to handle one connection, which means that you have a 1:1 relationship between connections and threads and are, therefore, limited by the number of threads you can create in the JVM.

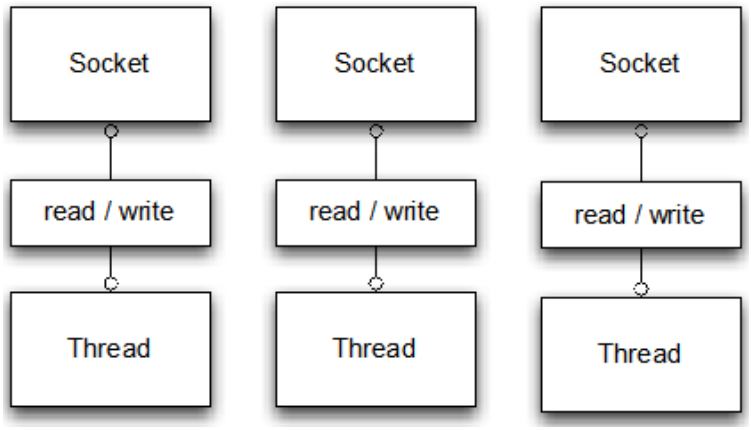


Figure 1.2 Blocking IO

In contrast, figure 1.3 shows how blocking IO lets you use a selector to handle multiple connections.

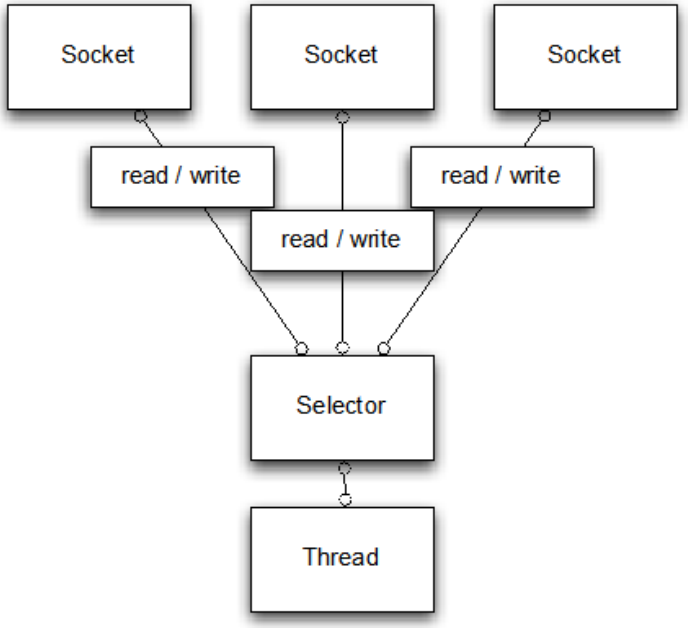


Figure 1.3 Non-blocking IO

Keeping these figures in mind, let's dive deeper into blocking IO and non-blocking IO. I'll use a simple echo server to demonstrate the difference between the IO and NIO. An echo server accepts client requests and echos (returns) the data that it receives from the clients.

1.3.1 EchoServer based on blocking IO

This first version of EchoServer, based on blocking IO, is probably the most common way to write network-related applications for two main reasons: the blocking IO API has been around since early versions of Java, and it's relatively easy to use.

Unless you run into scalability issues, the blocking IO isn't generally a problem. The following listing shows the EchoServer implementation.

Listing 1.4 EchoServer v1: IO

```
public class PlainEchoServer {

    public void serve(int port) throws IOException {
        final ServerSocket socket = new ServerSocket(port);              #1
        try {
            while (true) {
                final Socket clientSocket = socket.accept();             #2
                System.out.println("Accepted connection from " +
clientSocket);

                new Thread(new Runnable() {                              #3
                    @Override
                    public void run() {
                        try {

                            BufferedReader reader = new BufferedReader(
                                    new
InputStreamReader(clientSocket.getInputStream()));
                            PrintWriter writer = new PrintWriter(clientSocket
                                    .getOutputStream(), true);
                            while(true) {                                #4
                                writer.println(reader.readLine());
                                writer.flush();
                            }
                        } catch (IOException e) {
                            e.printStackTrace();
                            try {
                                clientSocket.close();
                            } catch (IOException ex) {
                                // ignore on close
                            }
                        }
                    }
                }).start();                                              #5
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

#1 Bind server to port

#2 Block until new client connection is accepted
#3 Create new thread to handle client connection
#4 Read data from client and write it back
#5 Start thread

This listing should look familiar to you if you've ever written a network application in Java. But let's pause and think about it: what problems may be present with this sort of "design"?

Let's revisit the following lines:

```
final Socket clientSocket = socket.accept();
new Thread(new Runnable() {
    @Override
    public void run() {
        ...
    }
}).start();
```

One thread is needed for each new client connection that is served. You may argue that we could make use of a thread pool to get rid of the overhead of creating the threads, but that would only help for a while. The fundamental problem still remains: the number of concurrent clients that can be served is limited to the number of threads that you can have "alive" at the same time. When your application needs to handle thousands of concurrent clients, that's a big problem.

This problem isn't present when using NIO as I'll demonstrate in the next version of EchoServer. But first, it's important to understand a couple of key NIO concepts.

1.3.2 Non-blocking IO basics

Java 7 introduced a new NIO API known as NIO.2, but you can use either NIO or NIO.2. Although the new API is also asynchronous, it's different from the original NIO implementation in both its API and implementation. Still, the APIs aren't completely different, and both share common features. For example, both implementations use an abstraction called a `ByteBuffer` as a data container.

BYTEBUFFER

A `ByteBuffer` is fundamental to both NIO APIs and, indeed, to Netty. A `ByteBuffer` can either be allocated on the heap or directly, which means it's stored outside of the `Heap-Space`. Usually, using a direct buffer is faster when passing it to the channel, but the allocation/deallocation costs are higher. In both cases, the API for a `ByteBuffer` is the same, which provides a unified way of accessing and manipulating data. A `ByteBuffer` allows the same data to be easily shared between `ByteBuffer` instances without the need to do any memory copying. It further allows slicing and other operations to limit the data that's visible.

Slicing

Slicing a ByteBuffer allows to create a new ByteBuffer that share the same data as the intial ByteBuffer but only expose a sub-region of it. This is useful to minimize memory copies while still only allow access to a part of the data

Typical uses of the `ByteBuffer` include the following:

- Writing data to the `ByteBuffer`
- Calling `ByteBuffer.flip()` to switch from write-mode to reading-mode.
- Reading data out of the `ByteBuffer`
- Calling either `ByteBuffer.clear()` or `ByteBuffer.compact()`

When you write data to the `ByteBuffer`, it keeps track of the amount of data you've written by updating the position of the write index in the buffer; this can also be done manually.

When you're ready to read the data, you call `Bytebuffer.flip()` to switch from writing-mode to reading-mode. Calling `ByteBuffer.flip()` sets the limit of the `ByteBuffer` to the current position and then update its position to 0. This way, you can read all of the data in the `ByteBuffer`.

To write to the `ByteBuffer` again, to switch back to writing-mode and then call either of the following methods:

- `ByteBuffer.clear()`—Clears the entire `ByteBuffer`
- `Bytebuffer.compact()`—Clears only the data that's already been read via memory copying

`ByteBuffer.compact()` moves all unread data to the beginning of the `ByteBuffer` and adjusts the position. The following listing shows how you would typically use a `ByteBuffer`.

Listing 1.5 Working with a `ByteBuffer`

```
Channel inChannel = ....;
ByteBuffer buf = ByteBuffer.allocate(48);

int bytesRead = -1;
do {
    bytesRead = inChannel.read(buf);                          #1
    if (bytesRead != -1) {
    buf.flip();                                               #2
    while(buf.hasRemaining()){
        System.out.print((char) buf.get());                       #3
    }
    buf.clear();                                              #4
    }
} while (bytesRead != -1);
inChannel.close();
```

#1Read data from the Channel to the ByteBuffer
#2 Make buffer ready for read

#3 Read the bytes in the ByteBuffer; every get() operation updates the position by 1
#4 Make the ByteBuffer ready for writing again

Now that you understand how a `ByteBuffer` is used, let's move on to the concept of selectors.

WORKING WITH NIO SELECTORS

The NIO API, which is still the most widely used of the two NIO APIs, uses a selector-based approach to handle network events and data.

A channel represents a connection to an entity capable of performing IO operations such as a file or a socket.

A selector is a NIO component that determines if one or more channels are ready for reading and/or writing, thus a single select selector can be used to handle multiple connections, alleviating the need for the thread-per-connection model you saw in the blocking IO EchoServer example.

To use selectors, you typically complete the following steps.

1. Create one or more selectors to which opened channels (sockets) can be registered.
2. When a channel is registered, you specify which events you're interested in listening in.

   The four available events (or Ops/operations) are:

- `OP_ACCEPT`—Operation-set bit for socket-accept operations
- `OP_CONNECT`—Operation-set bit for socket-connect operations
- `OP_READ`—Operation-set bit for read operations
- `OP_WRITE`—Operation-set bit for write operations

3. When channels are registered, you call the `Selector.select()` method to block until one of these events occurs.
4. When the method unblocks, you can obtain all of the `SelectionKey` instances (which hold the reference to the registered channel and to selected Ops) and do something.

   What exactly you do depends on which operation is ready. A `SelectedKey` can include more than one operation at any given time.

To see how this works, let's implement a non-blocking version of EchoServer. You'll get an opportunity to work with both of the NIO implementations in more detail. You'll also see that the `ByteBuffer` is essential to both.

1.3.3 EchoServer based on NIO

As shown in the following listing, this version of the EchoServer uses the asynchronous NIO API, which allows you to serve thousands of concurrent clients with one thread!

Listing 1.6 EchoServer v2: NIO

```
public class PlainNioEchoServer {
    public void serve(int port) throws IOException {
        System.out.println("Listening for connections on port " + port);
```

```
        ServerSocketChannel serverChannel = ServerSocketChannel.open();
        ServerSocket ss = serverChannel.socket();
        InetSocketAddress address = new InetSocketAddress(port);
        ss.bind(address);                                                    #1
        serverChannel.configureBlocking(false);
Selector  selector = Selector.open();
        serverChannel.register(selector, SelectionKey.OP_ACCEPT);            #2


        while (true) {
            try {
                selector.select();                                           #3
            } catch (IOException ex) {
                ex.printStackTrace();
                // handle in a proper way
                break;
            }

            Set readyKeys = selector.selectedKeys();                         #4
            Iterator iterator = readyKeys.iterator();
            while (iterator.hasNext()) {
                SelectionKey key = (SelectionKey) iterator.next();
                iterator.remove();                                           #5
                try {
                    if (key.isAcceptable()) {
                        ServerSocketChannel server = (ServerSocketChannel)
key.channel();
                        SocketChannel client = server.accept();              #6
                        System.out.println("Accepted connection from " +
client);
                        client.configureBlocking(false);
                        client.register(selector, SelectionKey.OP_WRITE |
SelectionKey.OP_READ, ByteBuffer.allocate(100));                             #7
                    }
                    if (key.isReadable()) {                                  #8
                        SocketChannel client = (SocketChannel) key.channel();
                        ByteBuffer output = (ByteBuffer) key.attachment();
                        client.read(output);                                 #9
                    }
                    if (key.isWritable()) {                                 #10
                        SocketChannel client = (SocketChannel) key.channel();
                        ByteBuffer output = (ByteBuffer) key.attachment();
                        output.flip();
                        client.write(output);                               #11
                        output.compact();
                    }
                } catch (IOException ex) {
                    key.cancel();
                    try {
                        key.channel().close();
                    } catch (IOException cex) {
                    }
                }
            }
        }
    }
}
```

#1 Bind server to port
#2 Register the channel with the selector to be interested in new Client connections that get accepted
#3 Block until something is selected
#4 Get all SelectedKey instances
#5 Remove the SelectedKey from the iterator
#6 Accept the client connection
#7 Register connection to selector and set ByteBuffer
#8 Check for SelectedKey for read
#9 Read data to ByteBuffer
#10 Check for SelectedKey for write
#11 Write data from ByteBuffer to channel

This example is more complex than the previous version of EchoServer. This complexity is a trade off; asynchronous code is typically more complicated than its synchronous counterpart.

Semantically, the original NIO and the new NIO.2 API are similar, but their implementations are different. We'll take a look the differences next, and we'll implement version 3 of EchoServer.

1.3.4 EchoServer based on NIO.2

Unlike the original NIO implementation, NIO.2 allows you to issue IO operations and provide what is called a completion handler (`CompletionHandler` class). This completion handler gets executed after the operation completes. Therefore, execution of the completion handler is driven by the underlying system and the implementation is hidden from the developer. It also guarantees that only one `CompletionHandler` is executed for channel at the same time. This approach helps to simplify the code because it removes the complexity that comes with multithreaded execution.

The major difference between the original NIO and NIO.2 is that you don't have to check whether an event accours on the Channel and then trigger some action. In NIO.2 you can just trigger an IO operation and register a completion handler with it, this handler will then get notified once the operation complates. This removes the need to create your own application logic to check for completion, which itself results in unnecessary processing.

Now let's see how the same asynchronous EchoServer would be implemented with NIO2,implementation as shown in the next listing.

Listing 1.7 EchoServer v3: NIO.2

```
public class PlainNio2EchoServer {
    public void serve(int port) throws IOException {
        System.out.println("Listening for connections on port " + port);
        final AsynchronousServerSocketChannel serverChannel =
AsynchronousServerSocketChannel.open();
        InetSocketAddress address = new InetSocketAddress(port);
        serverChannel.bind(address);                                    #1
        final CountDownLatch latch = new CountDownLatch(1);
        serverChannel.accept(null, new
CompletionHandler<AsynchronousSocketChannel, Object>() {                #2
            @Override
            public void completed(final AsynchronousSocketChannel channel,
Object attachment) {
```

```
                serverChannel.accept(null, this);                          #3
                ByteBuffer buffer = ByteBuffer.allocate(100);
                channel.read(buffer, buffer,
                        new EchoCompletionHandler(channel));               #4
            }

            @Override
            public void failed(Throwable throwable, Object attachment) {
                try {
                    serverChannel.close();                                 #5
                } catch (IOException e) {
                    // ingnore on close
                } finally {
                    latch.countDown();
                }
            }
        });
        try {
            latch.await();
        } catch (InterruptedException e) {
          Thread.currentThread().interrupt();
        }
    }

    private final class EchoCompletionHandler implements
CompletionHandler<Integer, ByteBuffer> {
        private final AsynchronousSocketChannel channel;
        EchoCompletionHandler(AsynchronousSocketChannel channel) {
            this.channel = channel;
        }

        @Override
        public void completed(Integer result, ByteBuffer buffer) {
            buffer.flip();
            channel.write(buffer, buffer, new CompletionHandler<Integer,
ByteBuffer>() {                                                            #6
                @Override
                public void completed(Integer result, ByteBuffer buffer) {
                    if (buffer.hasRemaining()) {
                        channel.write(buffer, buffer, this);               #7
                    } else {
                        buffer.compact();
                        channel.read(buffer, buffer,
                                EchoCompletionHandler.this);                #8
                    }
                }

                @Override
                public void failed(Throwable exc, ByteBuffer attachment) {
                    try {
                        channel.close();
                    } catch (IOException e) {
                        // ingnore on close
                    }
                }
            });
        }
```

```
            @Override
            public void failed(Throwable exc, ByteBuffer attachment) {
                try {
                    channel.close();
                } catch (IOException e) {
                    // ingnore on close
                }
            }
        }
    }
```

#1Bind Server to port
#2 Start to accept new Client connections. Once one is accepted the CompletionHandler will get called.
#3 Again accept new Client connections
#4 Trigger a read operation on the Channel, the given CompletionHandler will be notified once something was read
#5 Close the socket on error
#6 Trigger a write operation on the Channel, the given CompletionHandler will be notified once something was written
#7 Trigger again a write operation if something is left in the ByteBuffer
#8 Trigger a read operation on the Channel, the given CompletionHandler will be notified once something was read

At first glance, it may look like more code than what you had when using the previous NIO implementation. But notice that NIO.2 handles threading and the creation of the so-called event loop for you. This approach simplifies the code needed to build a multithreaded NIO application, even though it may not seem to be the case in this example. As the complexity of the application increases, the gains become more evident because you'll be producing cleaner code.

In the next section, we'll look at some known problems that exist in the JDK's NIO implementation.

1.4 NIO problems and how Netty comes to the rescue

In this section, we'll look at some problems and limitations of the Java NIO APIs and how Netty addresses these issues. Having the NIO packages of the JDK is a great step in the right direction, but users are limited in how they can use them. These issues are often the result of design choices that were made in the past and aren't easy to change now, others are defects.

1.4.1 Cross-platform and compatibility issues

NIO is low level and depends on how the operating system (OS) handles IO. So fulfilling the requirement of having a unified API exposed in Java, which works and behaves the same on all OSs, isn't trivial.

When using NIO you often find that your code works fine on Linux, for example, but has problems on Windows. My general advice, even if you're not using NIO, but it's more important if you are, is to test on all the OSs you want to support or use. Even if all the tests pass on your Linux workstation, be sure to verify it on the other OSs. If you don't verify, prepare to have fun later.

Ideal as NIO.2 may seem, it's only supported in Java 7, and if your application runs on Java 6, you may not be able to use it. Also, at the time of writing, there is no NIO.2 API for datagram channels (for UDP applications), so its usage is limited to TCP applications only.

Netty addresses this problem by providing a unified API, which allows the same semantics to work seamlessly on either Java 6 or 7. You don't have to worry about the underlying version, and you benefit from a simple and consistent API.

1.4.2 Extending ByteBuffer ... or not

As you saw previously, `ByteBuffer` is used as data container. Unfortunately, the JDK doesn't contain a `ByteBuffer` implementation that allows wrapping an array of `ByteBuffer` instances. This functionality is useful if you want to minimize memory copies. If you 'rethinking *I'll implement it myself*, don't waste your time; `ByteBuffer` has a private constructor, so it isn't possible to extend it.

Netty provides its own `ByteBuffer` implementation, which gets around this limitation and goes further by providing several other means of constructing, using, and manipulating a `ByteBuffer` all with a simpler API.

1.4.3 Scattering and gathering may leak

Many channel implementations support scattering and gathering. This feature allows writing to or reading from many `ByteBuffer` instances at the same time with better performance. Here the kernel/OS handles how these are written/read, which often gives you the best performance because the kernel/OS, being closer to the hardware, knows how to do it in the most efficient way.

Scattering/gathering is often used if you want to split data in different `ByteBuffer` instances to handle each buffer separately. For example, you may want to have the header in one `ByteBuffer` and the body in another.

Figure 1.4 shows how a scattering read is performed. You pass an array of `ByteBuffer` instances to the `ScatteringByteChannel` and the data gets scattered from the channel to the buffers.

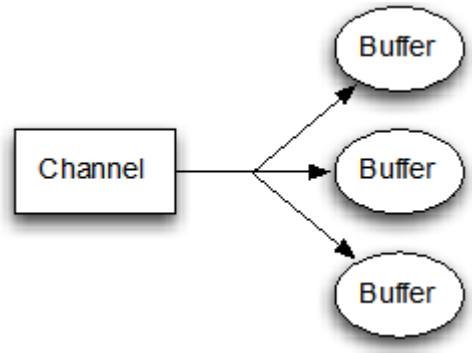


Figure 1.4 A scattering read from a channel

Gathering writes work in a similar way, but the data is written to the channel. You pass an array of `ByteBuffer` instances to the `GatheringByteChannel.write()` method and the data is gathered from the buffers to the channel.

Figure 1.5 illustrates the gathering write process.

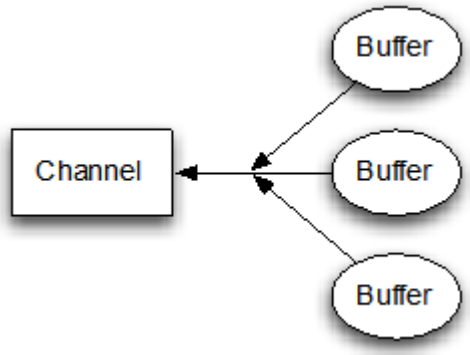


Figure 1.5 A gathering write to a channel

Unfortunately, this feature was broken in Java until a late Java 6 update and Java 7, and results in a memory leak, which causes an `OutOfMemoryError`. You need to be careful when you use scattering/gathering and be sure that you use the correct Java version on your production system.

You may ask, "Why not upgrade Java ?" I agree that this fixes the problem, but in reality it's often not feasible to upgrade, as your company may have restrictions on what version may be deployed on all the systems. Changing this may be painful or, in fact, not possible.

1.4.4 Squashing the famous epoll bug

On Linux-like OSs the selector makes use of the `epoll`– IO event notification facility. This is a high-performance technique in which the OS works asynchronously with the networking stack. Unfortunately, even today the "famous" `epoll`– bug can lead to an "invalid" state in the selector, resulting in 100% CPU-usage and spinning. The only way to recover is to recycle the old selector and transfer the previously registered `Channel` instances to the newly created `Selector`.

What happens here is that the `Selector.select()` method stops to block and returns immediately—even if there are no selected `SelectionKeys` present. This is against the contract, which is in the Javadocs of the `Selector.select()` method: `Selector.select()` must not unblock if nothing is selected.

NOTE For more details on the problem, see https://github.com/netty/netty/issues/327.

The range of solutions to this `epoll-` problem is limited, but Netty attempts to automatically detect and prevent it. The following listing is an example of the `epoll-` bug.

Listing 1.8 `Epoll-` bug in action

```
...
   while (true) {
        int selected = selector.select();                                    #1
        Set<SelectedKeys> readyKeys = selector.selectedKeys();
        Iterator iterator = readyKeys.iterator();                            #2
        while (iterator.hasNext()) {                                         #3
            ...
            ...                                                              #4
    }
}
...
```

#1Returns immediately and returns 0 as nothing was selected
#2 Obtains all SelectedKeys, Iterator is empty as nothing was selected
#3 Loops over SelectedKeys in Iterator, but never enters this block as nothing was selected
#4 Does the work in here

The effect of this code is that the while loop eats CPU:

```
...
while (true) {

}
...
```

The value will never be false, and the code keeps your CPU spinning and eats resources. This can have some undesirable side effects as it can consume all of your CPU, preventing any other CPU-bound work.

Figure 1.4 shows a Java process that takes up all of your CPU.

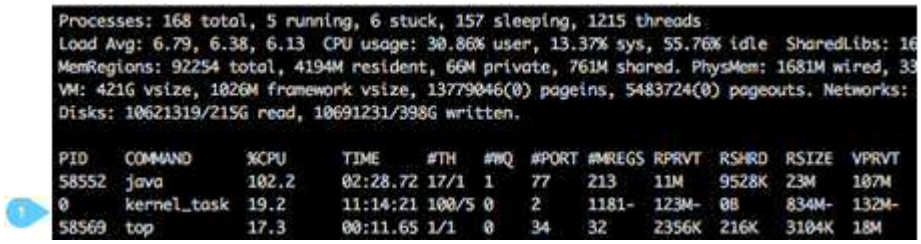


Figure 1.4 The `top` utility running in a Terminal window shows that java eats 100% CPU

#1The java command with the PID 58552 eats 102.2 % CPU

These are only a few of the possible problems you may see while using non-blocking IO. Unfortunately, even after years of development in this area, issues still need to be resolved; thankfully, Netty addresses them for you.

1.5 Summary

This chapter provided an overview of Netty's features, design and benefits. I discussed the difference between blocking and non-blocking processing to give you a fundamental understanding of the reasons to use a non-blocking framework.

You learned how to use the JDK API to write network code in both blocking and non-blocking modes. This included the new non-blocking API, which comes with JDK 7. After seeing the NIO APIs in action, it was also important to understand some of the known issues that you may run into. In fact, this is why so many people use Netty: to take care of workarounds and other JVM quirks.

In the next chapter, you'll learn the basics of the Netty API and programming model, and, finally, use Netty to write some useful code.

2

Your first Netty application

This chapter covers

- Getting the latest version of Netty
- Setting up the required environment to build and run the examples
- Creating a Netty client and server
- Intercepting and handling errors
- Building and running the Netty client and server

This chapter prepares you for the rest of this book by providing you with a gentle introduction to core Netty concepts. One such concept is learning how Netty allows you to intercept and handle exceptions, which is crucial when you're getting started and need help debugging problems. The chapter also introduces you to other core concepts, such as client and server bootstrapping and separation of concerns via channel handlers. To provide a foundation to build on as you progress through future chapters, you'll set up a client and server to communicate with each other using Netty. First, though, you need to set up your development environment.

2.1 Setting up the development environment

To set up a development environment, complete the following steps:

1. Install Java to compile and run the listed examples.

   Your operating system may come with a JDK; if so, you can skip this step. If you need to install Java, the latest version is available at http://java.com. Make sure you install the JDK (and not the JRE), as a JDK is needed for compiling the code. If you need the installation documentation, please refer to the Java website.

2. Download and install Apache Maven.

Maven is a dependency management tool, which makes dependency tracking easier. Download the archive from its project website at http://maven.apache.org. At the time of this book's publication, the most recent version is 3.0.5.

Make sure you download the correct archive for your operating system. For Windows, this is the .zip archive, for UNIX-like operating systems (such as Linux and Mac OSX), download the tar.bz2 archive.

After the file downloads, extract the archive to a folder of your choice, following the procedures specific to your operating system.

NOTE The Netty project uses Maven, and in keeping with this, all of our examples use Maven as well.

The following listing shows the Maven files that may be included on an OSX system.

Listing 2.1 Unpacking Maven archive

```
   Normans-MacBook-Pro:Apps norman$ tar xfvz ~/Downloads/apache-maven-3.0.5-
   bin.tar.gz
   Normans-MacBook-Pro:Apps norman$ tar xfvz ~/Downloads/apache-maven-3.0.5-
   bin.tar.gz
   x apache-maven-3.0.5/boot/plexus-classworlds-2.4.jar
   x apache-maven-3.0.5/lib/maven-embedder-3.0.5.jar
   x apache-maven-3.0.5/lib/maven-settings-3.0.5.jar
   x apache-maven-3.0.5/lib/plexus-utils-2.0.6.jar
   x apache-maven-3.0.5/lib/maven-core-3.0.5.jar
   x apache-maven-3.0.5/lib/maven-model-3.0.5.jar
   x apache-maven-3.0.5/lib/maven-settings-builder-3.0.5.jar
   x apache-maven-3.0.5/lib/plexus-interpolation-1.14.jar
   x apache-maven-3.0.5/lib/plexus-component-annotations-1.5.5.jar
   x apache-maven-3.0.5/lib/plexus-sec-dispatcher-1.3.jar
   x apache-maven-3.0.5/lib/plexus-cipher-1.7.jar
   x apache-maven-3.0.5/lib/maven-repository-metadata-3.0.5.jar
   x apache-maven-3.0.5/lib/maven-artifact-3.0.5.jar
   x apache-maven-3.0.5/lib/maven-plugin-api-3.0.5.jar
   ...
   ...
   ...
   x apache-maven-3.0.5/lib/ext/README.txt
Normans-MacBook-Pro:Apps norman$
```

After you unpack the archive, you may want to add it to your path so that you can type `mvn` to execute Maven. Otherwise, you'll need to use the full path to the executable every time you want to run it.

Again, how this is done varies depending on your operating system.

- For UNIX-like operating systems, you usually configure this in a .profile or .bashrc file:

```
Normans-MacBook-Pro:Apps norman$ vim ~/.profile
# Inside the .profile file
export PATH=$PATH:/Users/norman/Apps/apache-maven-3.0.5/bin
```

- Save the changes with the `:wq` command. On Windows, set the path through the System Preferences pane.

Maven uses its Standard Directory Layout for organizing projects. You can find out more about this layout on the Maven website. I'll focus only on the important directories for the examples here. Table 2.1 shows the main directories you'll be working with in your Maven projects.

Table 2.1 Trimmed-down Maven Standard Directory Layout

Folder	Description
`/{$projectname}`	Root folder of the project
`/{$projectname}/src/main/java`	Application source

Maven uses an instruction file named pom.xml, as shown in the following listing, to specify how a project is built. Place pom.xml in the root folder of the project.

Listing 2.2 Maven pom.xml

```
<?xml version="1.0" encoding="ISO-8859-15"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.manning.nettyinaction</groupId>
  <artifactId>netty-in-action</artifactId>
  <name>netty-in-action</name>
  <version>0.1-SNAPSHOT</version>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>2.5.1</version>
        <configuration>
          <optimize>true</optimize>
          <source>1.6</source>
          <target>1.6</target>
        </configuration>
      </plugin>
    </plugins>
  </build>

  <dependencies>
    <dependency>
      <groupId>io.netty</groupId>
      <artifactId>netty-all</artifactId>
```

```
        <version>4.0.0.Final</version>
      </dependency>
    </dependencies>
</project>
```

You should end up with a directory structure having a pom.xml file and an src/main/java directory. In pom.xml, each dependency tag is a container for a library that Maven automatically includes. Netty is the only one listed in this example, but you can have as many as you need.

2.2 Netty client and server overview

The goal of this section is to guide you through building a complete Netty client and server. Typically, you may only be interested in writing a server, such as an HTTP server where the client would be a browser. For this example, however, you'll get a much clearer view of the entire lifecycle if you implement both the client and server.

The big-picture view of a Netty application looks like figure 2.1.

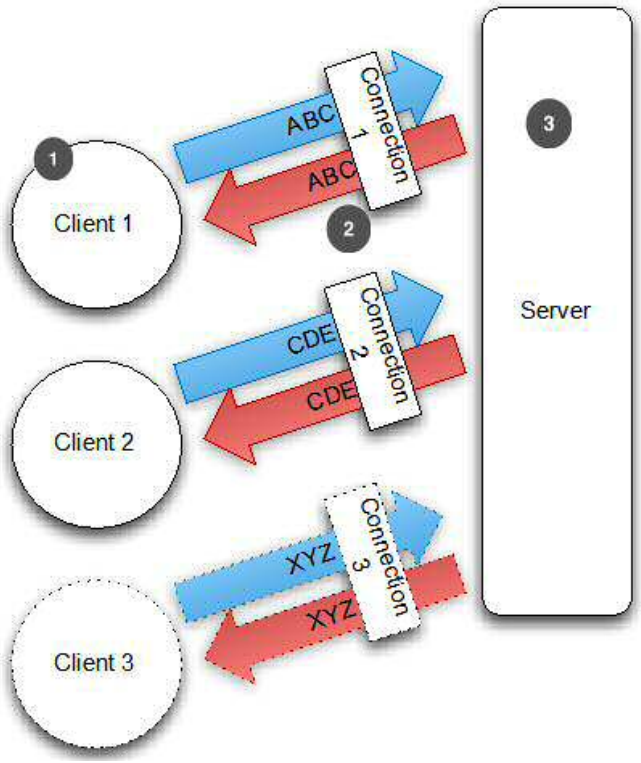


Figure 2.1 Application overview

#1 Client that connects to server
#2 Established connection to send/receive data
#3 Server that handles all connected clients

One thing that should be clear from looking at figure 2.1 is that the Netty server you'll write, automatically handles several concurrent clients. In theory, the only limits are the resources available on the system and any JDK limitations.

To make it easier to understand, imagine being in a valley or on a mountain and shouting something, then hearing what you shouted echoed back to you. In this scenario, you're the client and the mountain is the server. By going to the mountain, you make a connection. The act of shouting is analogous to a Netty client sending data to the server. Hearing your voice echoed back is the same as the Netty server returning the same data you sent. After you leave the mountain, you're disconnected, but you can return to reconnect to the server and send more data.

Although it's not typically the case for the same data to be echoed back to a client, this to and fro between the client and server is common. Examples later in the chapter will repeatedly demonstrate this as they become increasingly complex.

The next few sections walk you through the process of creating this echo client and server with Netty.

2.3 Writing an Echo server

Writing a Netty server consists of two main parts:

- *Bootstrapping*—Configure server features, such as the threading and port.
- *Implementing the server handler*—Build out the component that contains the business logic, which determines what should happen when a connection is made and data is received.

2.3.1 Bootstrapping the server

You bootstrap a server by creating an instance of the `ServerBootstrap` class. This instance is then configured, as shown in the following listing, to set options, such as the port, the threading model/event loop, and the server handler to handle the business logic (for this example, it echoes data back, but it can be quite complex).

Listing 2.3 Main class for the server

```
public class EchoServer {

   private final int port;
          public EchoServer(int port) {
        this.port = port;
    }

   public void start() throws Exception {
        EventLoopGroup group = new NioEventLoopGroup();
        try {
            ServerBootstrap b = new ServerBootstrap();                  #1
```

```
            b.group(group)                                                  #2
             .channel(NioServerSocketChannel.class)                         #2
             .localAddress(new InetSocketAddress(port))                     #2
             .childHandler(new ChannelInitializer<SocketChannel>() {        #3
                 @Override
                 public void initChannel(SocketChannel ch)
throws Exception {
                     ch.pipeline().addLast(
                             new EchoServerHandler());                      #4
                 }
             });

            ChannelFuture f = b.bind().sync();                              #5
            System.out.println(EchoServer.class.getName() +                 #6
                " started and listen on " + f.channel().localAddress());#7
            f.channel().closeFuture().sync();                               #8
        } finally {                                                         #9
            group.shutdownGracefully().sync();                              #10
        }
    }

    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();
    }
}
```

#1 Bootstraps the server
#2 Specifies NIO transport, local socket address
#3 Adds handler to channel pipeline
#4 Binds server, waits for server to close, and releases resources

This example may seem trivial, but it does everything the vanilla Java examples did in chapter 1 and more. To bootstrap the server, you first create a `ServerBootstrap` instance (#1). Because you're using the NIO transport, you specify the `NioEventLoopGroup` to accept new connections and handle accepted connections, specify the `NioServerSocketChannel` as the channel type, and you set the `InetSocketAddress` to which the server should bind so that it accepts new connections (#2).

Next, you specify the `ChannelHandler` to call when a connection is accepted, which creates a child channel (#3). A special type named `ChannelInitializer` is used here.

Even though the NIO examples in chapter 1 are scalable, they're prone to other issues. Threading, for example, isn't easy to get right, but Netty's design and abstraction encapsulates most of the threading work you'd need to do, through the use of `EventLoopGroup`, `SocketChannel`, and `ChannelInitializer`, each of which will be discussed in more detail in a later chapter.

The `ChannelPipeline` holds all of the different `ChannelHandlers` of a channel, so you add the previously written `EchoServerHandler` to the channel's `ChannelPipeline` (#4).

At (#5), you bind the server and then wait until the bind completes, the call to the "sync()" method will cause this to block until the server is bound. At #7 the application will wait until the server's channel closes (because we call sync() on the channel's close future). You can now shutdown the `EventLoopGroup` and release all resources, including all created threads(#10).

NIO is used for this example because it's currently the most used transport , and it's likely you'll use it too, But you can choose a different transport implementation. For example, if this example used the OIO transport, you'd specify `OioServerSocketChannel`. Netty's architecture, including what transports are, will be covered later.

Let's highlight the important things here:

- You create a `ServerBootstrap` instance to bootstrap the server and bind it later.
- You create and assign the `NioEventLoopGroup` instances to handle event processing, such as accepting new connections, receiving data, writing data, and so on.
- You specify the local `InetSocketAddress` to which the server binds.
- You set up a `childHandler` that executes for every accepted connection.
- After everything is set up, you call the `ServerBootstrap.bind()` method to bind the server.

2.3.2 Implementing the server/business logic

Netty uses the concept of futures and callbacks, discussed previously, coupled with a design that allows you to hook in and react to different event types. This is discussed in detail later, but for now let's focus on the data that's received. To do this, your channel handler must extend the `ChannelInboundHandlerAdapter` class and override the `messageReceived` method. This method is called every time messages are received, which in this case are bytes. This is where you place your logic to echo it back to the client, as shown in the following listing.

Listing 2.4 Channel handler for the server

```
@Sharable                                                          #1
public class EchoServerHandler extends
        ChannelInboundHandlerAdapter {

    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) {
        System.out.println("Server received: " + msg);

        ctx.write(msg)                                             #2
    }

    @Override
    public void channelReadComplete(ChannelHandlerContext ctx) {
        ctx.writeAndFlush(Unpooled.EMPTY_BUFFER)
                .addListener(ChannelFutureListener.CLOSE);         #3

    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx,
```

```
    Throwable cause) {
            cause.printStracktrace();                                    #4
            ctx.close();                                                 #5
        }
}
```

#1 Annotate with @Sharable to share between channels
#2 Write the received messages back . Be aware that this will not "flush" the messages to the remote peer yet.
#3 Flush all previous written messages (that are pending) to the remote peer, and close the channel after the operation is complete.
#4 Log exception
#5 Close channel on exception

Netty uses channel handlers to allow a greater separation of concerns, making it easy to add, update, or remove business logic as they evolve. The handler is straightforward and each of its methods can be overridden to "hook" into a part of the data lifecycle, but only the `channelRead` method is required to be overridden.

2.3.3 Intercepting exceptions

In addition to overriding the `channelRead` method, you may notice that also the `exceptionCaught` method was overriden. This is done to react to exceptions, or to any `Throwable` subtype. In this case, I log it and close the connection to the client, as the connection may be in an unknown state. Often this is what you do, but there may be scenarios where it's possible to recover from an error, so it's up to you to come up with a smart implementation. The important thing to note is that you should have at least one `ChannelHandler` that implements this method and so provides a way to handle all sorts of errors.

Netty's approach to intercepting exceptions makes it easier to handle errors that occur on different threads. Exceptions that were impossible to catch from separate threads are all fed through the same simple, centralized API.

There are many other `ChannelHandler` subtypes and implementations that you should be aware of if you plan to implement a real-world application or write a framework that uses Netty internally, but I'll talk about this later. For now, remember that `ChannelHandler` implementations will be called for different types of events and that you can implement or extend them to hook into the event lifecycle.

2.4 Writing an echo client

Now that all of the code is in place for the server, let's create a client to use it.

The client's role includes the following tasks:

- Connect to the server
- Writes data
- Waits for and receives the exact same data back from the server.
- Closes the connection

With this in mind, let's write the actual logic as you did before when you implemented the server.

2.4.1 Bootstrapping the client

As shown in the following listing, bootstrapping a client is similar to bootstrapping a server. The client bootstrap, however, accepts both a host and a port to which the client connects, as opposed to only the port.

Listing 2.5 Main class for the client

```
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();                          #1
            b.group(group)                                          #2
             .channel(NioSocketChannel.class)                       #3
             .remoteAddress(new InetSocketAddress(host, port))      #4
             .handler(new ChannelInitializer<SocketChannel>() {     #5
                 @Override
                 public void initChannel(SocketChannel ch)
throws Exception {
                     ch.pipeline().addLast(
                             new EchoClientHandler());              #6
                 }
             });

            ChannelFuture f = b.connect().sync();                   #7

            f.channel().closeFuture().sync();                       #8
        } finally {
            group.shutdownGracefully().sync();                      #9
        }
    }

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

        // Parse options.
        final String host = args[0];
        final int port = Integer.parseInt(args[1]);
```

```
            new EchoClient(host, port).start();
        }
    }
```

#1 Create bootstrap for client
#2 Specify EventLoopGroup to handle client events. NioEventLoopGroup is used, as the NIO-Transport should be used
#3 Specify channel type; use correct one for NIO-Transport
#4 Set InetSocketAddress to which client connects
#5 Specify ChannelHandler, using ChannelInitializer, called once connection established and channel created
#6 Add EchoClientHandler to ChannelPipeline that belongs to channel. ChannelPipeline holds all ChannelHandlers of channel
#7 Connect client to remote peer; wait until sync() completes connect completes
#8 Wait until ClientChannel closes. This will block.
#9 Shut down bootstrap and thread pools; release all resources

As before, the NIO transport is used here. It's worth mentioning that it doesn't matter what transport you use; you can use different transports in the client and server at the same time. It's possible to use the NIO transport on the server side and the OIO Transport on the client side. You'll learn more about which transport should be used in specific scenarios in chapter 4.

Let's highlight the important points of this section:

- A `Bootstrap` instance is created to bootstrap the client.
- The `NioEventLoopGroup` instance is created and assigned to handle the event processing, such as creating new connections, receiving data, writing data, and so on.
- The remote `InetSocketAddress` to which the client will connect is specified.
- A handler is set that will be executed once the connection is established.
- After everything is set up, the `ServerBootstrap.connect()` method is called to connect to the remote peer (the echo-server in our case).

2.4.2 Implementing the client logic

I'll keep this example simple, because any class used that hasn't yet been covered will be discussed in more detail in upcoming chapters.

As before, I write my own `SimpleChannelInboundHandlerAdapter` implementation to handle all the needed tasks, as shown in listing 2.6. This is done by overriding three methods that handle events that are of interest to us:

- `channelActive()`—Called after the connection to the server is established
- `channelRead0()`—Called after you receive data from the server
- `exceptionCaught()`—Called if any exception was raised during processing

Listing 2.6 Channel handler for the client

```
@Sharable                                                          #1
public class EchoClientHandler extends
        SimpleChannelInboundHandlerAdapter<ByteBuf> {
```

```
        @Override
        public void channelActive(ChannelHandlerContext ctx) {
            ctx.write(Unpooled.copiedBuffer("Netty rocks!",
                    CharsetUtil.UTF_8);                                    #2
        }

        @Override
        public void channelRead0(ChannelHandlerContext ctx,
                ByteBuf in) {
            System.out.println("Client received: " + ByteBufUtil
                .hexDump(in.readBytes(in.readableBytes())));  #4
        }

        @Override
        public void exceptionCaught(ChannelHandlerContext ctx,             #5
    Throwable cause) {
            cause.printStracktrace();
ctx.close();
        }
    }
```

#1 Annotate with @Sharable as it can be shared between channels
#2 Write message now that channel is connected
#3 Log received message as hexdump
#4 Log exception and close channel

As explained previously, you override three methods in this listing. All of these methods are needed to implement the scenario for which you're writing the application.

The `channelActive()` method is called once the connection is established. The logic there is simple. Once the connection is established, a sequence of bytes is sent to the server. The contents of the message doesn't matter; I used the encoded string of "Netty rocks!" Overriding this method ensures that something is written to the server as soon as possible.

Next, override the `channelRead0()` method. The method is called once data is received. Note that the bytes may be fragmented, which means that if the server writes 5 bytes it's not guaranteed that all 5 bytes will be received at once. For 5 bytes, the `channelRead0()` method could be called twice, for example. The first time it may be called with a ByteBuf that holds 3 bytes and the second time with a ByteBuf that holds 2 bytes. The only guarantee is that the bytes will be received in the same order as they're sent. But this is only true for TCP or other stream-orientated protocols.

The third method to override is `exceptionCaught()`. Use the same method as with the `EchoServerHandler` (listing 2.3). The `Throwable` is logged and the channel is closed, which means the connection to the server is closed.

You may ask yourself why we now used SimpleChannelInboundHandler and not ChannelInboundHandlerAdapter as it was used in the EchoServerHandler. The main reason for this is that with ChannelInboundHandlerAdapter you are responsible to „release" resources after you handled the received message. In case of `ByteBuf` this will be call `ByteBuf.release()`. With `SimpleChannelInboundHandler` this is not the case as

it will release the message once the `channelRead0(...)` method completes. This is done by Netty to handle all messages that are implement `ReferenceCounted`.

But why not do the same in the `EchoServerHandler`? The reason for this is that we want to echo back the message, which means we can not release it yet as the write operation may completes after `channelRead(...)` returns (remember write is asynchronous). Once the write completes Netty will automatically release the message.

This is everything you need for the client side. Now it's time to test your code and see it in action.

2.5 Compiling and running the echo client and server

In this section, I'll go through the various steps needed to run your echo client and server. In large, complex projects, you often need to use a build tool, and as mentioned previously, all the examples in this book use Maven. You can compile the echo client and server with any other build tool (or even with the javac command). The most important thing is that the Netty jar needs to be on the class path because the code depends on it.

To compile the example, you have to do some preparation first, as the maven has it's own dependencies. The next section guides you through that process.

2.5.1 Compiling the server and client

Once everything is in place, it's easy to compile your application execute the `mvn` command from the root directory of the project (where pom.xml exists). The following listing shows the command output.

Listing 2.7 Compile the source

```
Normans-MacBook-Pro:netty-in-action norman$ mvn clean package
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
--
[INFO] Building netty-in-action 0.1-SNAPSHOT
[INFO] ------------------------------------------------------------------------
--
[INFO]
[INFO] --- maven-clean-plugin:2.4.1:clean (default-clean) @ netty-in-action -
--
[INFO] Deleting /Users/norman/Documents/workspace/netty-in-action/target
[INFO]
[INFO] --- maven-resources-plugin:2.4.3:resources (default-resources) @
netty-in-action ---
[WARNING] Using platform encoding (UTF-8 actually) to copy filtered
resources, i.e. build is platform dependent!
[INFO] skip non existing resourceDirectory
/Users/norman/Documents/workspace/netty-in-action/src/main/resources
[INFO]
[INFO] --- maven-compiler-plugin:2.5.1:compile (default-compile) @ netty-in-
```

```
action ---
[INFO] Compiling 4 source files to /Users/norman/Documents/workspace/netty-
in-action/target/classes
[INFO]
[INFO] --- maven-resources-plugin:2.4.3:testResources (default-testResources)
@ netty-in-action ---
[INFO] skip non existing resourceDirectory
/Users/norman/Documents/workspace/netty-in-action/src/test/resources
[INFO]
[INFO] --- maven-compiler-plugin:2.5.1:testCompile (default-testCompile) @
netty-in-action ---
[INFO] No sources to compile
[INFO]
[INFO] --- maven-surefire-plugin:2.7.2:test (default-test) @ netty-in-action
---
[INFO] No tests to run.
...
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
There are no tests to run.

Results :

Tests run: 0, Failures: 0, Errors: 0, Skipped: 0                        #1

[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ netty-in-action ---
[INFO] Building jar: /Users/norman/Documents/workspace/netty-in-
action/target/netty-in-action-0.1-SNAPSHOT.jar                          #2
[INFO] ------------------------------------------------------------------------
--
[INFO] BUILD SUCCESS                                                    #3
[INFO] ------------------------------------------------------------------------
--
[INFO] Total time: 4.344s
[INFO] Finished at: Mon Oct 22 09:49:28 CEST 2012
[INFO] Final Memory: 13M/118M
[INFO] ------------------------------------------------------------------------
--
Normans-MacBook-Pro:netty-in-action norman$
```

#1 Unit tests report ran, failed, errors, or skipped
#2 Java jar file created from echo client and server code and placed in location listed
#3 Compilation successful; errors would report "build failed"

Maven downloads all the libraries your code needs. In this case, only Netty is required, but for bigger projects there may be more dependencies.

After the process completes you'll have your final JAR file, which you'll run in the next section.

2.5.2 Running the server and client

So everything is compiled now and ready to be used. This is done via the `java` command. You'll need at least two console windows. The first console runs the server; the second runs the client.

The following listing shows how the server is started. Note that I include the Netty jar and the generated jar (which holds your compiled code) with the classpath. This is needed or you'd get a `ClassNotFoundException`.

Listing 2.8 Starting the server

```
Normans-MacBook-Pro:netty-in-action norman$ java -cp
~/.m2/repository/io/netty/netty-all/4.0.0.Final/netty-all-4.0.0.
Final.jar:target/netty-in-action-0.1-SNAPSHOT.jar
com.manning.nettyinaction.chapter2.EchoServer 8080


com.manning.nettyinaction.chapter2.EchoServer started and listen on
/0:0:0:0:0:0:0:0:8080
```

To stop the server, press Ctrl-C.

Now the server is started and ready to accept data, which it processes and echoes back to the client. To test this, I'll start up the client, which does the following:

- Connects to the server.
- Writes data to the server.
- Waits to receive data.
- Shuts down the server .

Again, use the `java` executable to start up the client, with the classpath set up correctly as shown in the following listing.

Listing 2.9 Starting the client

```
Normans-MacBook-Pro:netty-in-action norman$ java -cp
~/.m2/repository/io/netty/netty-all/4.0.0.Final/netty-all-
4.0.0.Final.jar:target/netty-in-action-0.1-SNAPSHOT.jar
com.manning.nettyinaction.chapter2.EchoClient 127.0.0.1 8080

Client received: 4e6574747920726f636b7321
```

Note that the client starts up and prints out a single log statement to STDOUT. This shows the HEX of the received data. After the HEX prints, it closes the connection and terminates itself without any further interaction.

Every time you start the client, you'll see one log statement in the console that the server is running in:

```
Server received: 4e6574747920726f636b7321
```

As you can see, the logged HEX matches what was logged on the client, which verifies that the application works as expected; the server writes exactly the same data to the client as it received.

The last scenario to test is what happens if the server isn't started but a client tries to connect to the server. First, stop the server with the Ctrl-C shortcut, as described previously. Once you're sure it's not running anymore, start the client again. The following listing shows the output of the client when the server is no longer running.

Listing 2.10 Exception in client

```
Normans-MacBook-Pro:netty-in-action norman$ java -cp
~/.m2/repository/io/netty/netty-all/4.0.0.Final/netty-all-
4.0.0.Final.jar:target/netty-in-action-0.1-SNAPSHOT.jar
com.manning.nettyinaction.chapter2.EchoClient 127.0.0.1 8080
java.net.ConnectException: Connection refused
     at sun.nio.ch.SocketChannelImpl.checkConnect(Native Method)
     at sun.nio.ch.SocketChannelImpl.finishConnect(SocketChannelImpl.java:692)
     at
io.netty.channel.socket.nio.NioSocketChannel.doFinishConnect(NioSocketChannel
.java:177)                                                    #1
     at
io.netty.channel.socket.nio.AbstractNioChannel$AbstractNioUnsafe.finishConnec
t(AbstractNioChannel.java:201)
     at
io.netty.channel.socket.nio.NioEventLoop.processSelectedKeys(NioEventLoop.jav
a:284)
     at io.netty.channel.socket.nio.NioEventLoop.run(NioEventLoop.java:211)
     at
io.netty.channel.SingleThreadEventExecutor$1.run(SingleThreadEventExecutor.ja
va:80)
     at java.lang.Thread.run(Thread.java:722)
Exception in thread "main" io.netty.channel.ChannelException:
java.net.ConnectException: Connection refused                 #2
     at
io.netty.channel.DefaultChannelFuture.rethrowIfFailed(DefaultChannelFuture.ja
va:226)
     at
io.netty.channel.DefaultChannelFuture.sync(DefaultChannelFuture.java:175)
     at
com.manning.nettyinaction.chapter2.EchoClient.start(EchoClient.java:37)
     at                                                       #3
com.manning.nettyinaction.chapter2.EchoClient.main(EchoClient.java:56)
Caused by: java.net.ConnectException: Connection refused
     at sun.nio.ch.SocketChannelImpl.checkConnect(Native Method)
     at sun.nio.ch.SocketChannelImpl.finishConnect(SocketChannelImpl.java:692)
     at
io.netty.channel.socket.nio.NioSocketChannel.doFinishConnect(NioSocketChannel
.java:177)
     at
io.netty.channel.socket.nio.AbstractNioChannel$AbstractNioUnsafe.finishConnec
t(AbstractNioChannel.java:201)
     at
io.netty.channel.socket.nio.NioEventLoop.processSelectedKeys(NioEventLoop.jav
a:284)
```

```
    at io.netty.channel.socket.nio.NioEventLoop.run(NioEventLoop.java:211)
    at
io.netty.channel.SingleThreadEventExecutor$1.run(SingleThreadEventExecutor.ja
va:80)
    at java.lang.Thread.run(Thread.java:722)
```

#1 Client connect fails; even on different thread Netty propagates exception
#2 Exception traceable back to client example
#3 Specific exception propagated and not wrapped. ConnectException could be handled differently in client handler to support features such as automatic reconnection

What happened? The client tried to connect to the server, which it expected to run on 127.0.0.1:8080. This failed (as expected) because the server was stopped previously, triggering a `java.net.ConnectException` on the client side. The exception then triggered the `exceptionCaught(…)` method of the `EchoClientHandler`. This prints out the stack trace and closes the channel, which is exactly how it was implemented in listing 2.3.

You've now essentially built a client and server which can handle concurrency in a simpler way. This setup is capable of scaling to several thousand concurrent users and handles far more messages per second than a vanilla Java can handle. In later chapters, you'll' see and appreciate why and how Netty's approach makes scaling and threading easier. You'll also see that by allowing for a separation of concerns and hooking into the data lifecycle, Netty creates an extensible environment in which it's easy to implement improvements and changes as the business requirements change.

2.6 Summary

In this chapter you got an introduction to Netty by implementing a basic server and client. You learned how to compile the code that is used for examples in this book and how to get and install all the tools needed. These tools are used later in the more advanced examples included in this book.

This chapter also provided you with a simple application that shows how applications are broken down when using Netty to develop them. Finally, you learned how to intercept exceptions and handle them, both on the client and the server.

3

Netty from the ground up

In this chapter we're going to take a 10K feet view of Netty. Doing so will help you understand how Netty's components fit together and how they are useful to you.

There are some things without which an application would never work (there are others but in terms of Netty these are perhaps the most common/important ones you'll come across).

- `Bootstrap` or `ServerBootstrap`
- `EventLoop`
- `EventLoopGroup`
- `ChannelPipeline`
- `Channel`
- `Future` or `ChannelFuture`
- `ChannelInitializer`
- `ChannelHandler`

That is the purpose of this chapter, to introduce all of these concepts in preparation for the rest of the book. Instead of explaining what these are separately we'll describe how they all work together in order to help you build a mental picture of how the pieces fit in.

3.1 Netty Crash Course

Before we begin, it'll be useful if you get a rough idea of the general structure of a Netty application (Both client and servers have a similar structure).

A Netty application begins with one of the Bootstrap classes, a Bootstrap is a construct Netty provides that makes it easy for you to configure how Netty should setup or "bootstrap" the application.

In order to allow multiple protocols and various ways of processing data, Netty has what are called handlers. Handlers, as their names suggest are designed to handle a specific "event"

or sets of events in Netty. An event is a very generic way of describing this because you can have a handler, which converts an object to bytes or visa-versa or you can have a handler, which is notified about Exceptions that are thrown during processing and handle them.

One very common type you'll be writing is an implementation of the `ChannelInboundHandler`. The `ChannelInboundHandler` receives messages which you can process and decide what to do with it. You may also write/flush data out from inside an `ChannelInboundHandler` when your application needs to provide a response. In other words, the business logic of your application typically lives in a `ChannelInboundHandler`.

Business logic

The business logic represent this part of your program which does actual some work with the data that is received via the network after it was transformed to some format which is well understood. This could be for example be some kind of storing it into a database or something similar. What exactly it does depends completly on the application you are writiing, but represent a „core part" of your application.

When Netty connects a client or binds a server it needs to know how to process messages that are sent or received. This is also done via different types of handlers but to configure these handlers Netty has what is known as an `ChannelInitializer`. The role of the `ChannelInitializer` is to add `ChannelHandler` implementations to what's called the `ChannelPipeline`. As you send and receive messages, these handlers will determine what happens to the messages. An `ChannelInitializer` is also itself a `ChannelHandler` which automatically removes itself from the `ChannelPipeline` after it has added the other handlers.

All Netty applications are based on what is called a `ChannelPipeline`. The `ChannelPipeline` is closely related to what's known as the `EventLoop` and `EventLoopGroup` because all three of them are related to events or event handling.

An `EventLoops` purpose in the application is to process IO operations for a `Channel`. A single `EventLoop` will typically handle events for multiple `Channels`. The `EventLoopGroup` itself may contain more then one `EventLoop` and can be used to obtain an `EventLoop`.

A Channel is a representation of a socket connection or some component capable of performing IO operations, hence why it is managed by the `EventLoop` whose job it is to process IO.

All IO operations in Netty are performed asynchronously. So when you connect to a host for example, this is done asynchronously by default. The same is true when you write/send a message. This means the operation may not be performed directly but picked up later for execution. Because of this you can't know if an operation was successful or not after it returns, but need to be able to check later for success or have some kind of ways to register a listener which is notified. To rectify this, Netty uses Futures and `ChannelFutures`. This future can be

used to register a listener, which will be notified when an operation has either failed or completed successfully.

ChannelFuture – what's that ?

Like stated before all IO operations are asynchronous in Netty and thus may complete later and not directly when you execute a method. This is where ChannelFuture comes into play. A ChannelFuture is a special java.util.concurrent.Future, which allos you to register ChannnelFutureListeners tot he ChannelFuture. Those ChannelFutureListeners will get notified once the operation (which was triggered by the method call) is complete.

So basically a ChannelFuture is a placeholder for a result of an operation that is executed in the future. When exactly it is executed depends on many facts and is not easy to „say". The only thing you can be sure of is that it will be executed and all operations that return a ChannelFuture and belong to the same Channel will be executed in the correct order, which is the same order as you executed the methods.

As you go through the rest of this chapter and this book you'll find these concepts are mentioned very often. But they're not overly complex so if you come across a more detailed explanation of each which seems complex, just skip back to this section and re-read this for the simplified version.

In the rest of this chapter, we're going to expand upon everything we just mentioned.

3.2 Channels, Events and Input/Output (IO)

Netty is a non-blocking, event driven, networking framework. In reality what this means for you is that Netty uses threads to process IO events. For those who are familiar with multi-threaded programming, you might be inclined to think that you need to synchronize your code. This isn't true, figure 3.1 shows why the design Netty uses to ensure no synchronization is required on your part to process Netty events.

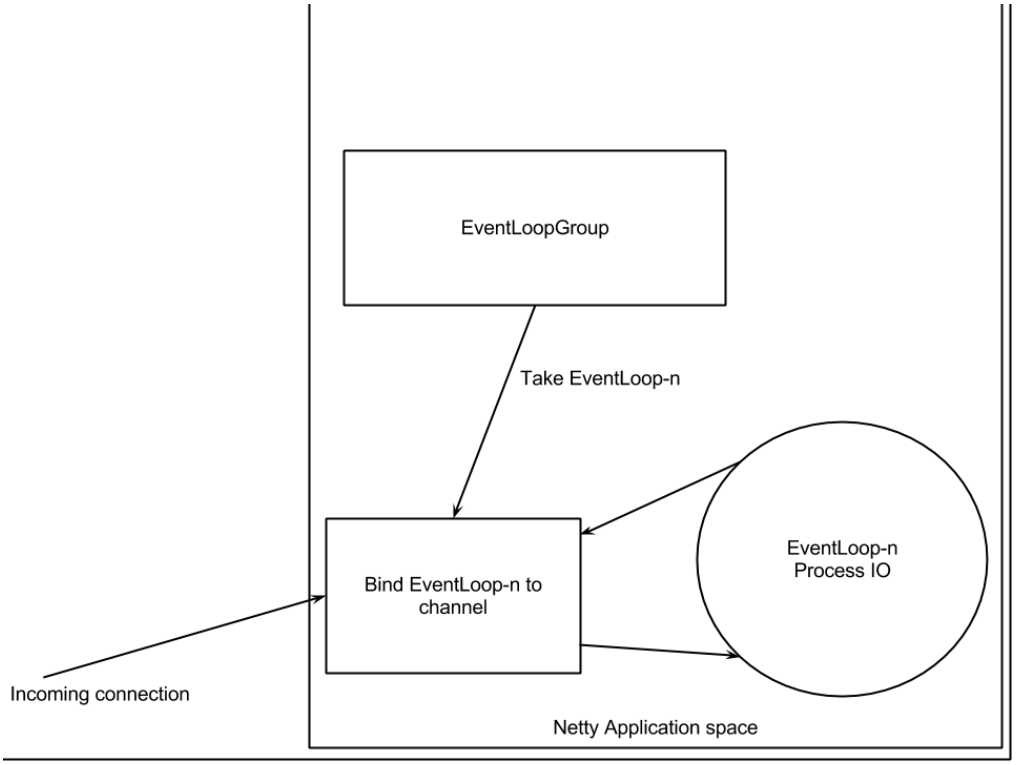


Figure 3.1 – Diagram showing an `EventLoopGroup` and a `Channel` being bound to a single `EventLoop` permanently

Figure 3.1 shows that Netty has these `EventLoopGroups`, those groups in turn have one or more `EventLoops`. Think of `EventLoops` as the threads that perform the actual work for a channel.

EventLoop Thread relationship

The EventLoop is always bound to a single Thread that never changed during it's life time.

When a channel is registered, Netty "binds" that channel to a single `EventLoop` (and so to a single thread) for the lifetime of that `Channel`. This is why your application doesn't need to synchronize on Netty IO operations because all IO for a given `Channel` will always be performed by the same thread.

To help explain this, figure 3.2 shows the relationship of `EventLoops` to `EventLoopGroups`.

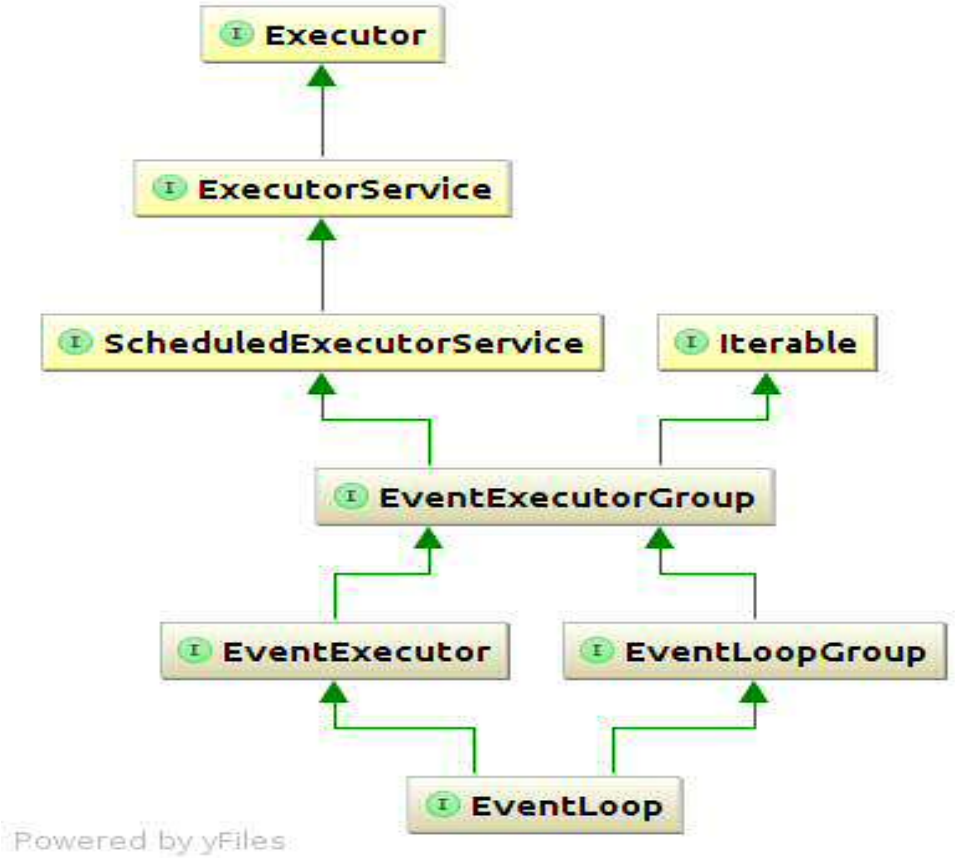


Figure 3.2 – Inheritance relationship for `EventLoops` and `EventLoopGroups`

The relationship between an `EventLoop` and an `EventLoopGroup` may not be immediately intuitive, because we've said that an `EventLoopGroup` contains one or more `EventLoop` but this diagram shows that in fact, an `EventLoop` passes the "is-a" `EventLoopGroup` test, i.e. an `EventLoop` is an `EventLoopGroup`. This means wherever you can pass in an `EventLoopGroup` you can also just use a specific `EventLoop`.

3.3 Bootstrapping: What and Why

Bootstrapping in Netty is the process by which you configure your Netty application. You use a bootstrap when you need to connect a client to some host and port, or bind a server to a given

port. As the previous statement implies, there are two types of Bootstraps, one typically used for clients, but is also used for `DatagramChannel` (simply called `Bootstrap`) and one for servers (aptly named `ServerBootstrap`). Regardless of which protocol or protocols your application uses, the only thing that determines which bootstrap you use is whether you're trying to create a client or a server.

There are several similarities between the two types of bootstraps, in fact, there are more similarities than there are differences. Table 3.1 shows some of the key similarities and differences between the two.

Table 3.1 Similarities and differences between the two types of Bootstraps

Similarities	Bootstrap	ServerBootstrap
Responsible for	Connects to a remote host and port	Binds to local port
Number of EventLoopGroups	1	2

Groups, transports and handlers are covered separately later in this chapter, so we'll only take a look t the key differences between the types of Bootstraps. The first difference should be obvious; "`ServerBoostrap`" binds to a port since servers must listen for connections while a "`Bootstrap`" is used for client applications or `DatagramChannel`. With the "`Bootstrap`" class you typically call `connect()` but you can also call `bind()` and then connect later using the `Channel` that is included in the `ChannelFuture` returned from `bind()`.

The second difference is perhaps the most important. Client bootstraps/applications use a single `EventLoopGroup` whilst `ServerBootstrap` uses 2 (which in fact can be the same instance). It may not be immediately obvious as to why this would be but it's a good reason. A `ServerBootstrap` can be thought to have two sets of channels. The first set containing a single `ServerChannel` representing the server's own socket which has been bound to a local port. And the second set containing all the `Channel` representing the connections, which the server has accepted. Figure 3.3 tries to help visualize what this looks like.

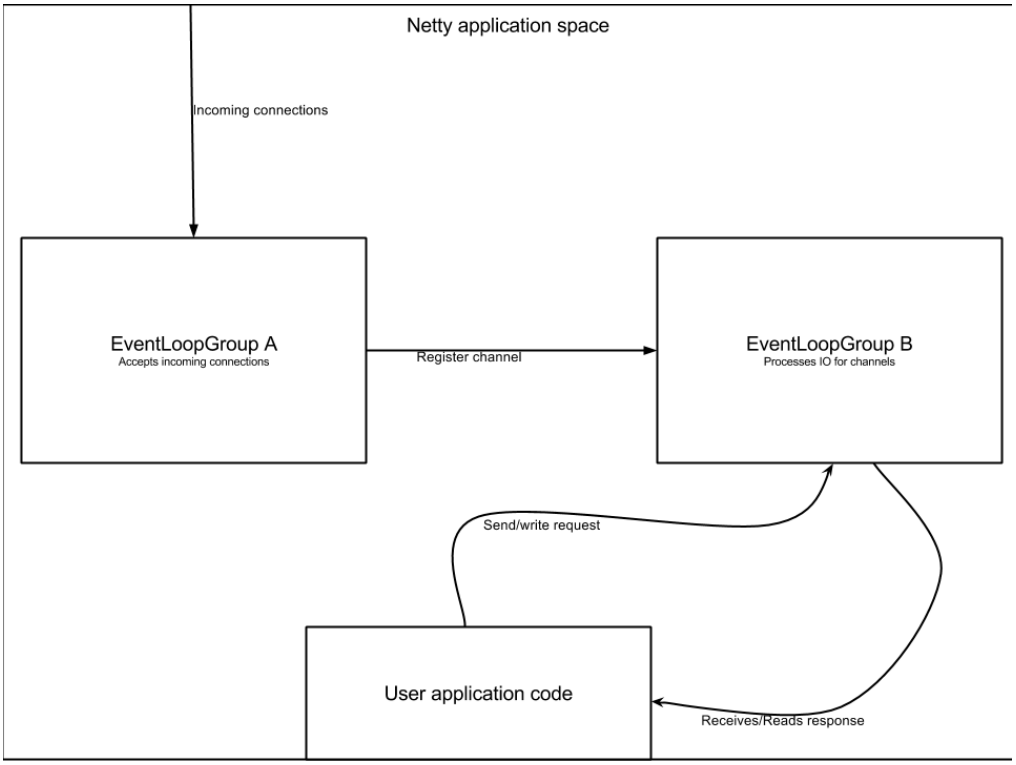


Figure 3.3 - Represents the distinction between server and client groups

In figure 3.3, `EventLoopGroup` A's only purpose is to accept connections and hand them over to `EventLoopGroup` B. The reason Netty has the ability to use two distinct set of groups is because in situations where an application is accepting an extremely high volume of connections, a single `EventLoopGroup` would become a bottleneck if the EventLoop is busy with handle the already accepted connections and thus is not be able to accept new ones within a reasonable time. The end result would be that some connections timeout. By having two `EventGroups`, all connections can be accepted, even under extremely high load because the `EventLoops` (and so underlying threads) accepting connections are not shared with those processing the already accepted connections.

EventLoopGroup and EventLoop

The EventLoopGroup may contain more then one EventLoop, but if so depends on configuration. Each Channel will have one EventLoop „bind" to it once it was created which will never change. Because a EventLooopGroup contains naturally less EventLoop's then the

number of Channels you have many Channels will share the same EventLoop. This means keep th EventLoop to busy in one Channel will disallow to process the other Channels that are bound to the same EventLoop. This is one of the reasons why you MUST NOT block the EventLoop in all cases.

Figure 3.4 shows how this changes if you decide to configure your Netty server using the same instance of `EventLoopGroup` twice.

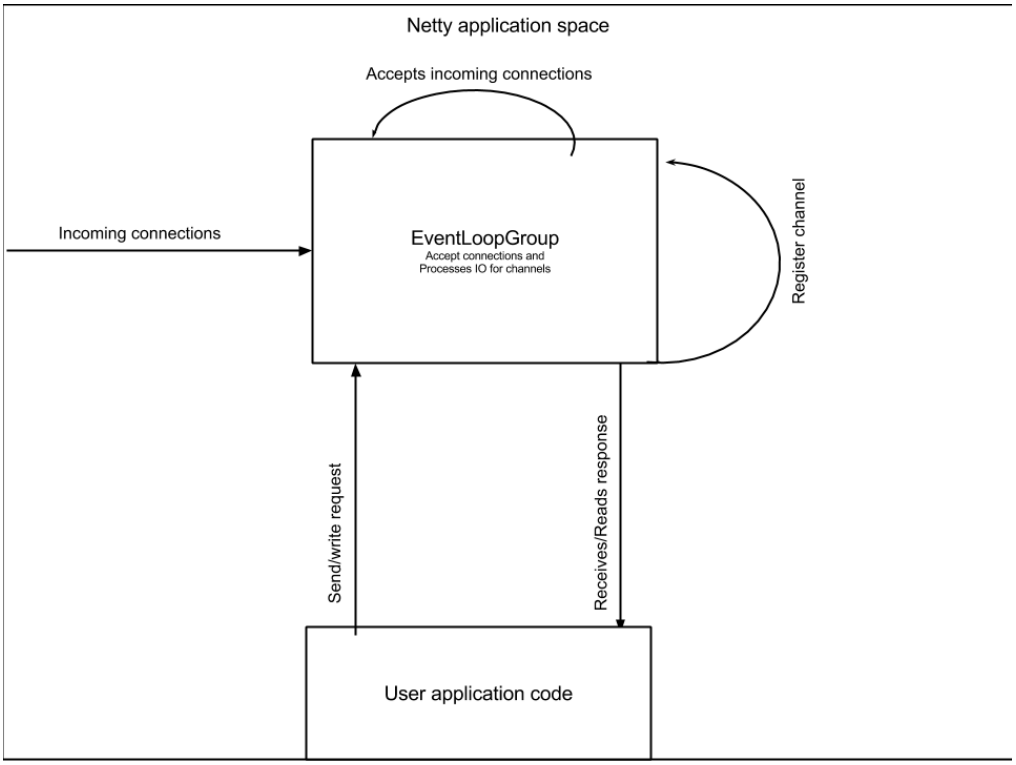


Figure 3.4 – Netty server configured with only 1 `EventLoopGroup`

Netty allows the same `EventLoopGroup` to be used for processing IO and accepting connections. In practice this works well for many applications. This case is represented in figure 3.4 where a single group, both accepts connections and processes IO.

In the next section we'll get into discussing how and when Netty performs IO operations.

3.4 Channel Handlers and Data Flow

We need to take a look at what happens to your data when you send or receive it. Re-call at the beginning of this chapter we mentioned that Netty has this concept of a handler. To understand what happens to data when it is written or read, it is first essential to have some understanding of what handlers are. Handlers themselves depend upon the aforementioned `ChannelPipeline` to prescribe their order of execution. Thus, it is not possible to define certain aspects of a handler without defining the `ChannelPipeline` and in turn it is not possible to define certain aspects of the `ChannelPipeline` without defining `ChannelHandlers`. Needless to say we must start with one and proceed to the other defining each in terms of themselves and each other. The next subsection will introduce both `ChannelHandlers` and the `ChannelPipeline` in a way that makes this cyclic dependency negligible.

3.4.1 Piecing it together, ChannelPipeline and handlers

In many ways a Netty `ChannelHandler` is what your application deals with the most. Even when you don't realize it, if you're using a Netty application there is going to be at least one `ChannelHandler` involved somewhere. In other words, they are key to many things. So what exactly are they? It's not easy to give a definition of a `ChannelHandler` because they are so generic but a `ChannelHandler` can be thought of as any piece of code that processes data coming and going through the `ChannelPipeline`. Practically there is one parent interface defining a handler called `ChannelHandler`. From this a `ChannelInboundHandler` and `ChannelOutboundHandler` are derived as shown in figure 3.5.

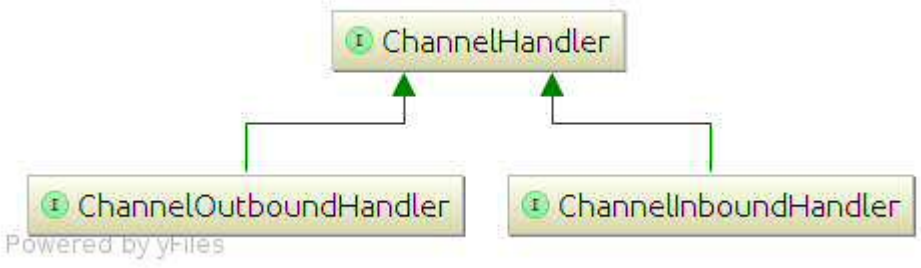


Figure 3.5 – Relationship between `ChannelHandler` and it's derivatives

It is easier if we explain `ChannelHandlers` in terms of data flow but it is important to remember that the examples used in this discussion are just that, examples. `ChannelHandlers` are applied in many other ways, as you'll discover going through the rest of this book.

Data flows in two directions within Netty, as figure 3.5 shows there is a clear distinction between inbound (`ChannelInboundHandler`) and outbound (`ChannelOutboundHandler`)

handlers. Data is said to be outbound if the expected flow is from the user application to the remote peer. Conversely, data is inbound if it is coming from the remote peer to the user application.

In order for data to get from one end to another typically, one or more `ChannelHandler` would have manipulated the data in some way. These `ChannelHandlers` would have been added at the bootstrap phase of the application and the order in which they were added determines the order in which they would have manipulated the data.

This arrangement of `ChannelHandler` in a specific order effectively constitutes to what we've been referring to as the `ChannelPipeline`. In other words, the `ChannelPipeline` is an arrangement of a series of `ChannelHandler`. Each `ChannelHandler` performs its actions on the data (if it can handle it, for example inbound data can only be handled by `ChannelInboundHandlers`) then may pass the transformed data to the next `ChannelHandler` in the `ChannelPipeline`, until no more `ChannelHandler` remain.

ChannelHandler and Servlet similarities

In fact the design used within Netty is kind of similar to what is used in Servlets. A ChannelHandler may do some action on the data and then may pass it to the next ChannelHandler in the ChannelPipeline. Another often action is do not do any action at all and just pass the specific event to the next ChannelHandler in the ChannelPipeline. This next ChannelHandler may handle it then or just forward it to the next ChannelHandler again.

Figure 3.6 shows an example `ChannelPipeline` arrangement.

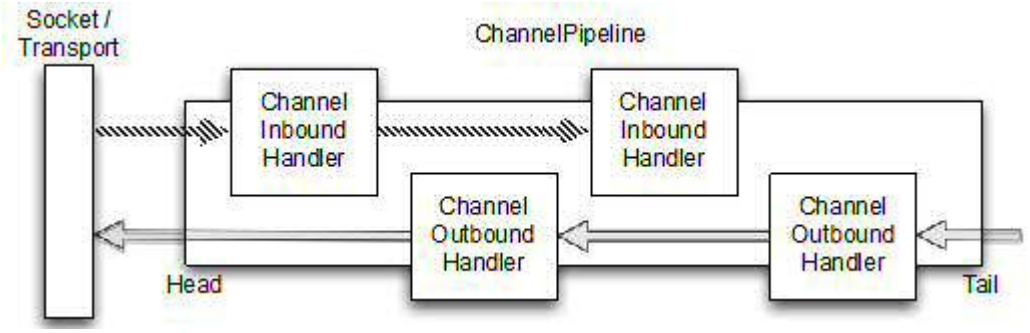


Figure 3.6 An example ChannelPipeline arragement

As figure 3.6 shows, both `ChannelInboundHandler` and `ChannelOutboundHandler` can be mixed into the same `ChannelPipeline`.

In this ChannelPipeline, if a message is read or any other inbound event, it will start from the head of the ChannelPipeline and be passed to the first ChannelInboundHandler. This ChannelInboundHandler may process the event and / or pass it to the next ChannelInboundHandler in the ChannelPipeline. Once there is no more ChannelnboundHandler in the ChannelPipeline it will hit the tail of the ChannelPipeline, which means no more processing will be done.

The inverse is also true, any outbound event (e,g, writes) will start from the tail of the ChannelPipeline and be passed to the "last" ChannelOutboundHandler in the ChannelPipeline. Here it acts the same as a ChannelInboundHandler, which means it may process and / or pass the event to the next ChannelOutboundHandler in the ChannelPipeline. The difference is that the next ChannelOutboundHandler is in fact the "previous" one as the outbound events flow from the tail to the head of the ChannelPipeline. Once there are no more ChannelOutboundHandler to which the event can be passed it will hit the actual transport (which may be a network socket) and so trigger some operation on it. Which may for example be a write operation.

ChannelInboundHandler and ChannelOutboundHandler base classes

An event can be forward to the next ChannelInbound or prev ChannelOutboundHandler in the ChannelPipeline by using the ChanneHandlerContext passed in to each method. Because this is what you usually want for events that you are not interested in Netty provides abstract base classes called ChannelInboundHandlerAdapter and ChannelOutboundHandlerAdapter. Each of this provide implementations for each method and just pass the event to the next/prev handler in the ChannelPipeline by call the corresponding method on the ChannelHandlerContext. You can then override the method in question to actually do the handling of you need to.

You might be wondering, so if outbound and inbound operations are distinct, how does this work when handlers are mixed in the same ChannelPipeline? Refer back to figure 3.4, remember that inbound and outbound handlers have a different interface that both extend ChannelHandler. This means that Netty can skip any handler that is not of a particular type and hence not able to handle a given operation. So in the case of an outbound event, ChannelInboundHandler will be skipped because Netty knows if each handler is and implementation of ChannelInboundHandler or ChannelOutboundHandler.

Once a ChannelHandler is added to a ChannelPipeline it also gets what's called a ChannelHandlerContext. Typically it is safe to get a reference to this object and keep it around. This is not true when a datagram protocol is used such as UDP. This object can later be used to obtain the underlying channel but is typically kept because you use it to write/send messages. This means there are two ways of sending messages in Netty. You can write directly to the channel or write to the ChannelHandlerContext object. The main difference between

the two is that writing to the channel directly causes the message to start from the tail of the `ChannelPipeline` where as writing to the context object causes the message to start from the next handler in the `ChannelPipeline`.

3.5 Encoders, Decoders and Domain Logic: A Closer Look at Handlers

As we said before, there are many different types of handlers. What each one does is dependent on which base class they inherit from. Netty provides a series of "Adapter" classes which makes things a bit easier. It does this because while in the pipeline, each handler is responsible for forwarding Netty events on to the next handler in the `ChannelPipeline`. With the *Adapter classes (and sub-classes) this is automatically done for you so you only need to override the methods/events you're interested in. Beside the *Adapter classes there are also others that extend those and provide extra functionalities like help to easy encode / decode message.

Adapter classes

There are a few adapter classes which allows you to write your ChannelHandlers in an easy way. Whenever you want to write your own ChannelHandler I suggest you to extend one of the adapter classes or one of the encoder/decoder classes (which in fact extend one of the adapter classes). Netty comes with the following adapters:

- ChannelHandlerAdapter
- ChannelInboundHandlerAdapter
- ChannelOutboundHandlerAdapter
- ChannelDuplexHandlerAdapter

Three `ChannelHandler` we want to take a particular look at are encoders, decoders and the `SimpleChannelInboundHandler<T>` (which is a sub-class of `ChannelInboundHandlerAdapter`).

3.5.1 Encoders, decoders

When you send or receive a message with Netty it must be converted from one form to another. If the message is being received it must be converted from bytes to a Java object (decoded by some kind of decoder). If the message is being sent it must be converted from a Java object to bytes (encoded by some type of encoder). This conversion will always happen when sending data over the network, byte-message or message-bytes, because you can only transfer bytes across a network.

There are various types of base classes for encoders and decoders, depending on what you want to do. For example, your application may use Netty in a way that doesn't require the message to be converted to bytes immediately so instead the message is converted to another

type of message. An encoder is still used but a different base class exists for that. To figure out which base class is applicable there is a little convention that can be used to know the name of the base class. In general base classes will have a name similar to "`ByteToMessageDecoder`" or "`MessageToByteEncoder`". Or in case of a specialized type you may find something like "`ProtobufEncoder`" and "`ProtobufDecoder`" which are used to support Google's protocol buffers.

Strictly speaking other handlers could do what encoders and decoders do but re-call we said that there are different adapter classes depending on what you wanted to do? In the case of decoders there is a `ChannelInboundHandlerAdapter` or `ChannelInboundHandler` which all decoders extend or implement. The "`channelRead`" method/event is overridden, this method is called by each message that is read from the inbound channel. The overridden `channelRead` method will then call the "decode" method of each decoder and forward the decoded message to the next `ChannelInboundHandler` in the `ChannelPipeline` via the `ChannelHandlerContext.fireChannelRead(decodedMessage)` method.

A similar thing happens when you send messages, except an encoder converts the message to bytes and those bytes are forwarded to the next `ChannelOutboundHandler`.

3.5.2 Domain logic

Perhaps the most common handler your application will have to deal with is the one which receives the decoded message and allows your application to apply some domain logic to the message. To create a handler like this, your application only needs to extend the base class called `SimpleChannelInboundHandler<T>`, where T is the type of message your handler can process. It is in this handler where your application obtains a reference to the `ChannelHandlerContext` by overriding one of the methods from the base class, all of them accept the `ChannelHandlerContext` as a parameter which you can then store as a field in the class.

This handler's main method of concern is the "`channelRead0(ChannelHandlerContext,T)`" method. Whenever Netty invokes this method, the object T is the message, which your application can then process. How you process the message is entirely up to you and the needs of the application. One thing to note while processing messages is that, even though there are multiple threads typically processing IO in Netty, your application should try not to block the IO thread as this could lead to performance issues in some high throughout environments.

Blocking operations

As said before you MUST NOT block the IO Thread at all. This means doing blocking operations within your ChannelHandler is problematic. Lucky enough there is a solution for this. Netty allows to specify an EventExecutorGroup when adding ChannelHandlers tot he ChannelPipeline. This EventExecutorGroup will then be used to obtain an EventExecutor and this EventExecutor will execute all the methods of the ChannelHandler. The EventExecutor

here will use a different Thread then the one that is used for the IO and thus free up the EventLoop.

3.6 Summary

This chapter presented many Netty concepts in a way that is meant to give you an idea of how all the pieces fit together. It does this because the following chapters discuss each of these topics at length in their individual chapters and it may not always be clear how the topic being discussed relates to everything else. You've just seen several key topics glanced over in a single chapter including

- `Bootstrap` or `ServerBootstrap`
- `EventLoop`
- `EventLoopGroup`
- `ChannelPipeline`
- `Channel`
- `Future` or `ChannelFuture`
- `ChannelInitializer`
- `ChannelHandler` and its sub-types

In the subsequent chapters you'll meet all of these topics again as they are covered in more detail.

4 Transports

This chapter covers

- Transports
- NIO, OIO, Local, Embedded
- Use-cases
- APIs

One of the most important tasks of a network application is transferring data. This can be done differently depending on the kind of transport used, but what gets transferred is always the same: bytes over the wire. Transports help abstract how the data is transferred. All you need to know is that you have bytes to send and receive. Nothing more, nothing less.

If you've ever worked with a Java-provided way of network programming you've probably come across a situation when you wanted to switch from blocking transports to nonblocking transports or vice versa. This switch isn't easily possible, because it uses different Java interfaces and classes to handle blocking versus nonblocking.

Netty offers a unified API on top of its transport implementation, which makes this situation easier. You'll be able to keep your code as generic as possible, and not depend on some implementation-dependent API. There won't be a need to refactor your whole code base because you need to move from one transport to another. If you ever needed to do so while using the plain network API that comes with the JDK, you already know how massive such changes can be. Don't waste your time with this boring stuff; spend it on something more productive.

This chapter shows you what the unified API looks like and how to use it. I'll compare it to the API that comes with the JDK and show you why Netty makes it easier to work with it. It will also explain the various transport implementations that are bundled with Netty and which is preferred for each use case. After gathering this information, you'll be able to choose the best option for your specific application, giving you the best result for your use case.

There's no other experience needed other than with Java itself. Having experience with network frameworks or network programming can help but isn't needed.

Let's see how transports work in a real-world situation.

4.1 Case study: transport migration

To give you an idea how transports work, I'll start with a simple application which does nothing but accept client connections and write "Hi!" to the client. After that's done, it disconnects the client. I won't get into the details of this specific implementation as it's only an example.

4.1.1 Using I/O and NIO without Netty

To start this case study I'll implement the application without Netty. The following listing shows the code using a blocking input/output (I/O).

Listing 4.1 Blocking networking without Netty

```
public class PlainOioServer {

    public void serve(int port) throws IOException {
        final ServerSocket socket = new ServerSocket(port);              #1
        try {
            while (true) {
                final Socket clientSocket = socket.accept();             #2
                System.out.println("Accepted connection from " +
clientSocket);

                new Thread(new Runnable() {                              #3
                    @Override
                    public void run() {
                        OutputStream out;
                        try {
                            out = clientSocket.getOutputStream();

out.write("Hi!\r\n".getBytes(Charset.forName("UTF-8")));                 #4
                            out.flush();
                            clientSocket.close();                        #5

                        } catch (IOException e) {
                            e.printStackTrace();
                            try {
                                clientSocket.close();
                            } catch (IOException ex) {
                                // ignore on close
                            }
                        }
                    }
                }).start();                                              #6
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

#1 Bind server to port

#2 Accept connection
#3 Create new thread to handle connection
#4 Write message to connected client
#5 Close connection once message written and flushed
#6 Start thread to begin handling

This works fine, but after some time you notice that the blocking handling doesn't scale enough for your use case. You want to use asynchronous networking to handle all the concurrent connections, but the problem is that the API is completely different. You start to rewrite your application as shown in the following listing.

Listing 4.2 Asynchronous networking without Netty

```
public class PlainNioServer {

    public void serve(int port) throws IOException {
        System.out.println("Listening for connections on port " + port);
        ServerSocketChannel serverChannel;
        Selector selector;

        serverChannel = ServerSocketChannel.open();
        ServerSocket ss = serverChannel.socket();
        InetSocketAddress address = new InetSocketAddress(port);
        ss.bind(address);                                                    #1
        serverChannel.configureBlocking(false);
        selector = Selector.open();                                          #2
        serverChannel.register(selector, SelectionKey.OP_ACCEPT);            #3
        final ByteBuffer msg = ByteBuffer.wrap("Hi!\r\n".getBytes());

        while (true) {

            try {
                selector.select();                                           #4
            } catch (IOException ex) {
                ex.printStackTrace();
                // handle in a proper way
                break;
            }

            Set<SelectedKey> readyKeys = selector.selectedKeys();            #5
            Iterator<SelectedKey> iterator = readyKeys.iterator();
            while (iterator.hasNext()) {
                SelectionKey key = iterator.next();
                iterator.remove();
                try {
                    if (key.isAcceptable()) {                                #6
                        ServerSocketChannel server = (ServerSocketChannel)
key.channel();
                        SocketChannel client = server.accept();
                        System.out.println("Accepted connection from " +
client);
                        client.configureBlocking(false);
                        client.register(selector, SelectionKey.OP_WRITE |
SelectionKey.OP_READ, msg.duplicate());                                  #7
                    }
```

```
                    if (key.isWritable()) {                                  #8
                        SocketChannel client = (SocketChannel) key.channel();
                        ByteBuffer buffer = (ByteBuffer) key.attachment();
                        while (buffer.hasRemaining()) {
                            if (client.write(buffer) == 0) {                 #9
                                break;
                            }
                        }
                        client.close();                                      #10
                    }
                } catch (IOException ex) {
                    key.cancel();
                    try {
                        key.channel().close();
                    } catch (IOException cex) {
                    }
                }
            }
        }
    }
}
```

#1 Bind server to port
#2 Open selector that handles channels
#3 Register erverSocket to selector and specify that it is interested in new accepted clients
#4 Wait for new events that are ready for process. This will block until something happens
#5 Obtain all SelectionKey instances that received events
#6 Check if event was because new client ready to get accepted
#7 Accept client and register it to selector
#8 Check if event was because socket is ready to write data
#9 Write data to connected client. This may not write all the data if the network is saturated. If so it will pick up the not-written data and write it once the network is writable again.
#10 Close connection

As you can see, the code is completely different, even if it does exactly the same thing. And this is only a simple application. Consider the work it would take to port something more complex.

With this in mind, let's implement the same application with Netty.

4.1.2 Using I/O and NIO with Netty

Start by writing a blocking version of the application, but this time use Netty as a network framework, as shown in the following listing.

Listing 4.3 Blocking networking with Netty

```
public class NettyOioServer {

    public void server(int port) throws Exception {
        final ByteBuf buf = Unpooled.unreleaseableBuffer(
           Unpooled.copiedBuffer("Hi!\r\n", Charset.forName("UTF-8")));
        EventLoopGroup group = new OioEventLoopGroup();
        try {
            ServerBootstrap b = new ServerBootstrap();                       #1
            b.group(group)                                                   #2
             .channel(OioServerSocketChannel.class)
```

```
                .localAddress(new InetSocketAddress(port))
                .childHandler(new ChannelInitializer<SocketChannel>() {      #3
                    @Override
                    public void initChannel(SocketChannel ch)
                        throws Exception {
                        ch.pipeline().addLast(
                                new ChannelInboundHandlerAdapter() {         #4
                            @Override
                            public void channelActive(
                                ChannelHandlerContext ctx) throws Exception {
                                ctx.write(buf.duplicate())
                                  .addListener(ChannelFutureListener.CLOSE);#5
                            }
                        });
                    }
                });
               ChannelFuture f = b.bind().sync();                            #6
               f.channel().closeFuture().sync();
            } finally {
               group.shutdownGracefully().sync();                            #7
            }
        }
    }
```

#1 Create ServerBootstrap to allow bootstrap to server instance
#2 Use OioEventLoopGroup Ito allow blocking mode (Old-IO)
#3 Specify ChannelInitializer that will be called for each accepted connection
#4 Add ChannelHandler to intercept events and allow to react on them
#5 Write message to client and add ChannelFutureListener to close connection once message written
#6 Bind server to accept connections
#7 Release all resources

You may notice that although the code itself is compact, it does exactly the same thing as the code in listing 4.1. But that's only one of the advantages.

Let's modify the code so that it works asynchronously.

4.1.3 Implementing asynchronous support

You'll see that the code in the following listing looks much the same as listing 4.3. A change of two lines of code is all that's needed to switch from blocking to asynchronous mode. You swap out the OIO transport for the NIO. The following listing shows the changed lines in bold.

Listing 4.4 Asynchronous networking with Netty

```
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();
        try {
            ServerBootstrap b = new ServerBootstrap();                       #1
            b.group(group)
             .channel(NioServerSocketChannel.class)
             .localAddress(new InetSocketAddress(port))
             .childHandler(new ChannelInitializer<SocketChannel>() {         #3
```

```
                    @Override
                    public void initChannel(SocketChannel ch)
                        throws Exception {
                        ch.pipeline().addLast(
                                new ChannelStateHandlerAdapter() {          #4
                            @Override
                            public void channelActive(
                                ChannelHandlerContext ctx) throws Exception {
                                ctx.write(buf.duplicate())
                                  .addListener(ChannelFutureListener.CLOSE);#5
                            }

                            @Override
                            public void inboundBufferUpdated(
                                ChannelHandHandlerContext ctx)
                                Throws Exception {
                                czx.fireInboundBufferUpdated();
                            }

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

#1 Create ServerBootstrap to allow bootstrap to server
#2 Use NioEventLoopGroup forl nonblocking mode
#3 Specify ChannelInitializer called for each accepted connection
#4 Add ChannelHandler to intercept events and allow to react on them
#5 Write message to client and add ChannelFutureListener to close connection once message written
#6 Bind server to accept connections
#7 Release all resources

Because Netty exposes the same API for every transport implementation, it doesn't matter what implementation you use. Netty exposes its operations through the `Channel` interface and its `ChannelPipeline` and `ChannelHandler`.

Now that you've seen Netty in action, let's take a deeper look at the transport API.

4.2 Transport API

At the heart of the transport API is the channel interface, which is used for all of the outbound operations.

See the hierarchy of the `Channel` interface as shown in figure 4.1.

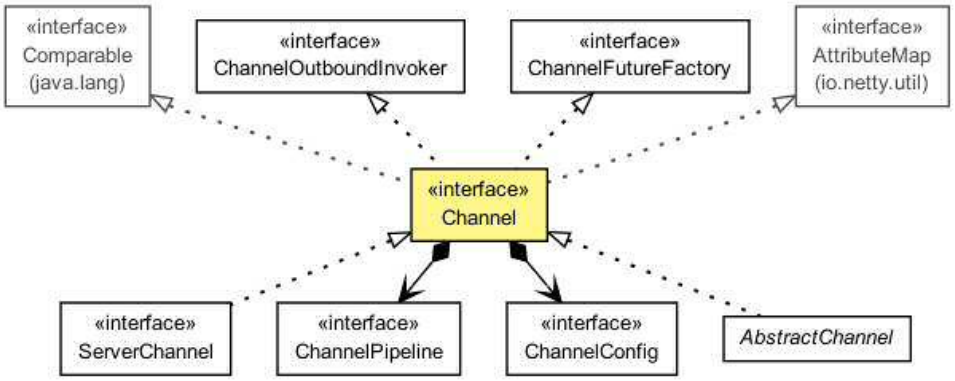


Figure 4.1 Channel interface hierarchy

As you can see in figure 4.1, a channel has a `ChannelPipeline` and a `ChannelConfig` assigned to it.

The `ChannelConfig` has the entire configuration settings stored for the channel and allows for updating them on the fly. Often transport has specific configuration settings that are possible only on the transport and not on other implementations. For this purpose it may expose a subtype of `ChannelConfig`. For more information, refer to the javadocs of the specific `ChannelConfig` implementation.

The `ChannelPipeline` holds all of the `ChannelHandler` instances that should be used for the inbound and outbound data that is passed through the channel. These `ChannelHandler` implementations allow you to react to state changes or transform data. This book includes a chapter on the details of `ChannelHandlers`, as these are one of the key concepts of Netty.

For now, I'll note that you can use `ChannelHandler` for these tasks:

- Transforming data from one format to another.
- Notifying you of exceptions.
- Notifying you when a Channel becomes active or inactive.
- Notifying you once a channel is registered/deregistered from an `EventLoop`.
- Notifying you about user-specific events.

These `ChannelHandler` instances are placed in the `ChannelPipeline`, where they execute one after the other. It's similar to a chain, which if you've used in servlets in the past, you may be familiar with. For more details on `ChannelHandler` topics, see chapter 6.

Channelpipeline

The ChannelPipeline implements the Intercepting Filter Pattern, which means you can chain different ChannelHandlers and intercept the data or events which go through the ChannelPipeline.

Think of it like UNIX pipes which allows to chain different Commands (where the ChannelHandler would be the Command here).

You can also modify the `ChannelPipeline` on the fly, which allows you to add/remove `ChannelHandler` instances whenever needed. This can be used to build highly flexible applications with Netty.

In addition to accessing the assigned `ChannelPipeline` and `ChannelConfig`, you can also operate on the `Channel` itself. The `Channel` provides many methods, but the most important ones are listed in table 4.1.

Table 4.1 Most important channel methods

Method name	Description
`eventLoop()`	Returns the `EventLoop` that is assigned to the channel
`pipeline()`	Returns the `ChannelPipeline` that is assigned to the channel
`isActive()`	Returns if the channel is active, which means it's connected to the remote peer
`localAddress()`	Returns the `SocketAddress` that is bound local
`remoteAddress()`	Returns the `SocketAddress` that is bound remote
`write()`	Writes data to the remote peer. This data is passed though the `ChannelPipeline`

You'll learn more later about how you can use all of these features. Remember for now that you'll always operate on the same interfaces, which gives you a high degree of flexibility and guards you from big refactoring once you want to try out a different transport implementation.

For writing data to the remote peer you'd call `Channel.write()` as shown in the following listing.

Listing 4.5 Writing to a channel

```
Channel channel = ...
ByteBuf buf = Unpooled.copiedBuffer(„your data“, CharsetUtil.UTF_8);     #1
ChannelFuture cf = channel.write(buf);                                    #2

cf.addListener(new ChannelFutureListener() {                              #3

    @Override
    public void operationComplete(ChannelFuture future) {
        if (future.isSuccess()) {                                         #4
            System.out.println(“Write successful“);
        } else {
            System.err.println(“Write error“);                            #5
            future.cause().printStacktrace();
        }
    }
});
```

#1 Create ByteBuf that holds data to write
#2 Write data
#3 Add ChannelFutureListener to get notified after write completes
#4 Write operation completes without error
#5 Write operation completed but because of error

Please note that Channel is thread-safe, which means it is safe to operate on it from different Threads. All it's methods are safe to use in a multi-thread enviroment. Because of this it's safe to store a reference to it in your application and use it once the need arises to write something to the remote peer, even when using many threads. The following listing shows a simple example of writing with multiple threads.

Listing 4.6 Using the channel from many threads

```
final Channel channel = ...
final ByteBuf buf = Unpooled.copiedBuffer(„your data“,
        CharsetUtil.UTF_8);                                               #1
Runnable writer = new Runnable() {                                        #2
    @Override
    public void run() {
        channel.write(buf.duplicate());
    }
};
Executor executor = Executors.newChachedThreadPool();                     #3

// write in one thread
executor.execute(writer);                                                 #4

// write in another thread
executor.execute(writer);                                                 #5
   ...
```

#1 Create ByteBuf that holds data to write
#2 Create Runnable which writes data to channel
#3 Obtain reference to the Executor which uses threads to execute tasks
#4 Hand over write task to executor for execution in thread
#5 Hand over another write task to executor for execution in thread

Also, this method guarantees that the messages are written in the same order as you passed them to the write method. For a complete reference of all methods, refer to the provided API documentation (javadocs).

Knowing the interfaces used is important, but it's also quite helpful to know what different transport implementations already ship with Netty. Chances are good that everything is already provided for you. In the next section I'll look at what implementations are provided and what their behaviors are.

4.3 Included transports

Netty already comes with a handful of transports that you can use. Not all of them support all protocols, which means the transport you want to use also depends on the underlying protocol that your application depends on. You'll learn more about which transport supports which protocol in this section.

Table 4.1 shows all of the transports that are included by default in Netty.

Table 4.1 Provided transports

Name	Package	Description
NIO	`io.netty.channel.socket.nio`	Uses the `java.nio.channels` package as a foundation and so uses a selector-based approach.
OIO	`io.netty.channel.socket.oio`	Uses the `java.net package` as a foundation and so uses blocking streams.
Local	`io.netty.channel.local`	A local transport that can be used to communicate in the VM via pipes.
Embedded	`io.netty.channel.embedded`	Embedded transport, which allows using `ChannelHandlers` without a real network based Transport. This can be quite useful for testing your `ChannelHandler` implementations.

Now let's go into more detail by first looking into the most-used transport implementation, the NIO transport.

4.3.1 NIO – Nonblocking I/O

The NIO transport is currently the most used. It provides a full asynchronous implementation of all I/O operations by using the selector-based approach that's included in Java since Java 1.4 and the NIO subsystem.

The idea is that a user can register to get notified once a channel's state changes. The possible changes are:

- A new `Channel` was accepted and is ready.
- A `Channel` connection was completed.
- A `Channel` has data received that is ready to be read.
- A `Channel` is able to send more data on the channel.

The implementation is then responsible for reacting to these state changes to reset them and to be notified once a state changes again. This is done with a thread that checks for updates and, if there are any, dispatches them accordingly.

Here it's possible to register to be notified for only one of the events and ignore the rest.

The exact bit-sets that are supported by the underlying selector are shown in table 4.2. These are defined in the `SelectionKey` class.

Table 4.2 Selection operation bit-set

Name	Description
`OP_ACCEPT`	Get notified once a new connection is accepted and a channel is created.
`OP_CONNECT`	Get notified once a connection attempt finishes.
`OP_READ`	Get notified once data is ready to be read out of the channel.
`OP_WRITE`	Get notified once it's possible to write more data to the channel. Most of the time this is possible, but it may not be because the OS socket buffer is completely filled. This usually happens when you write faster then the "remote peer" can handle it.

Netty's NIO transport uses this model internally to receive and send data, but exposes its own API to the user, which completely hides the internal implementation. As mentioned previously, that helps to expose only one unified API to the user, while hiding all of the internals. Figure 4.2 shows the process flow.

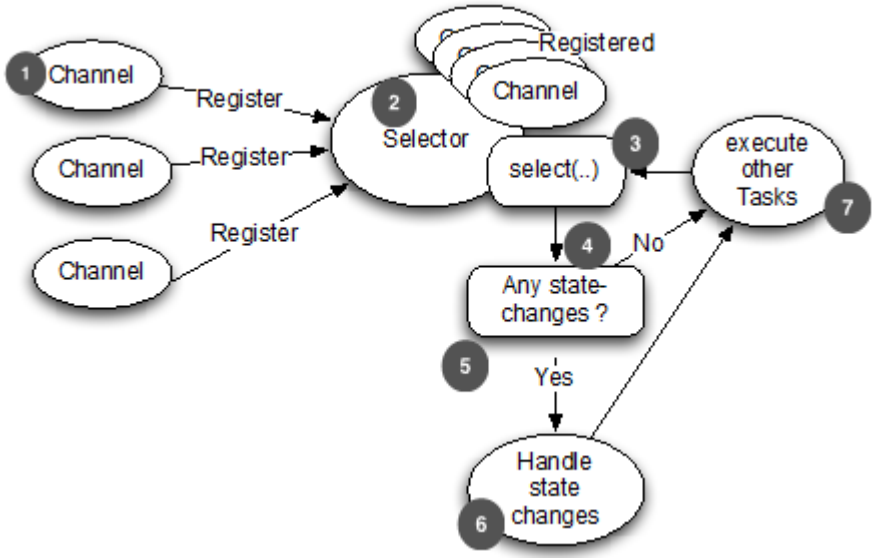


Figure 4.2 Selector logic

#1 New created channel that registers to selector
#2 Selector to handle state changes
#3 Already registered channels
#4 The Selector.select() method which blocks until new state changes received or given timeout elapsed
#5 Check if there were state changes
#6 Handle all state changes
#7 Execute other tasks in same thread in which selector operates

Because of the nature of this transport, it may come with a bit of latency when processing events, and so can have a lower throughput than the OIO transport. This is caused by the way the selector works, as it may take some time to be notified about state changes. I'm not talking about seconds of delay, only milliseconds. This may not sound like much of a delay, but it can add up if you try to use your network application in a network where gigabit speed is offered.

One feature that offers only the NIO transport at the moment is called "zero-file-copy". This feature allows you to quickly and efficiently transfer content from your file system. The feature provides a way to transfer the bytes from the file system to the network stack without copying the bytes from the kernel space to the user space.

Be aware that not all operation systems support this. Please refer to operating system's documentation to find out if it's supported. Also, be aware that you'll only be able to benefit from this if you don't use any encryption/compression of the data. Otherwise it will need to copy the bytes first to the user space to do the actual work, so only transferring the raw

content of a file makes use of this feature. What actually would work is to „pre-encrypt" a file before transfer it.

One application that can really make use of this is an FTP or HTTP server that downloads big files.

The next transport I'll discuss is the OIO transport, which provides a blocking transport.

4.3.2 OIO – OId blocking I/O

The OIO transport is a compromise in Netty. It builds on the known unified API but isn't asynchronous by nature because it uses the blocking java.net implementations under the hood. At first glance, this transport may not look useful to you, but it has its use cases. You'll see several use cases later, but in this section, we'll look at one specifically.

Suppose you need to port some legacy codethat uses many libraries that do blocking calls (such as database calls via jdbc). It may not be feasible to port the logic to not block. Instead, you could use the OIO transport in the short term and port it later to one of the pure asynchronous transports. Let's focus on how it works.

Because the OIO transport uses the java.net classes internally, it also uses the same logic that you may already be familiar with if you previously written network applications.

When using these classes, you usually have one thread that handles the acceptance of new sockets (server-side) and then creates a new thread for each accepted connection to serve the traffic over the socket. This is needed as every I/O operation on the socket may block at any time. If you share the same thread over more than one connection (socket), this could lead to a situation where blocking an operation could block all other sockets from doing their work.

Knowing that operations may block, you may start to wonder how Netty uses it while still providing the same way of building APIs. Here Netty makes use of the `SO_TIMEOUT` that you can set on a socket. This timeout specifies the maximum number of milliseconds to wait for an I/O operation to complete. If the operation doesn't complete within the specified timeout, a `SocketTimeoutException` is thrown. Netty catches this `SocketTimeoutException` and moves on with its work. Then on the next `EventLoop` run, it tries again. Unfortunately, this is the only way to do this and still confirm the inner working of Netty. The problem with this approach is that firing the `SocketTimeoutException` isn't free, as it needs to fill the `StrackTrace`, and so on.

Figure 4.3 shows the logic.

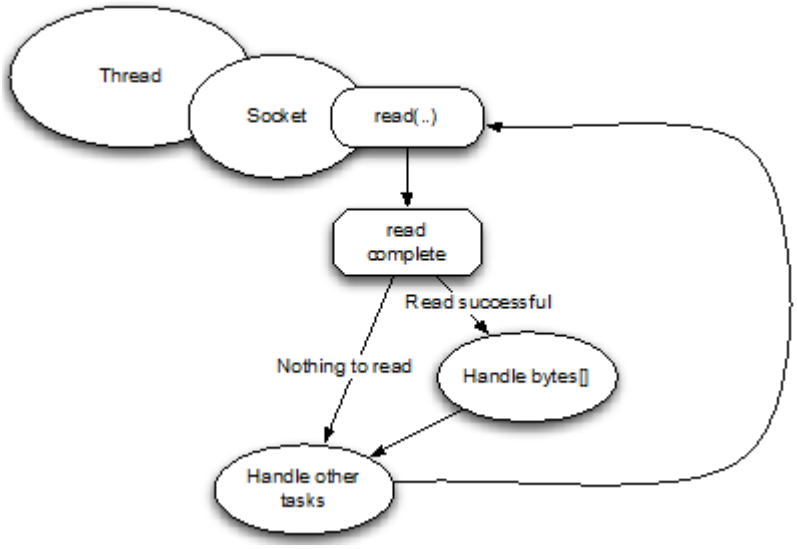


Figure 4.3 OIO-Processing logic

#1 Thread allocated to socket
#2 Socket connected to remote peer
#3 Read operation that may block
#4 Read complete
#5 Read completed and able to read bytes so handled them
#6 Execute other tasks submitted belonging to socket
#7 Try to read again

Now you know the two most used transports in Netty, but there are still others which we will cover in the next sections.

4.3.3 Local – In VM transport

Netty contains the so-called local transport. This transport implementation can be used to communicate within a VM and still use the same API you're used to. The transport is fully asynchronous as NIO .

Every Channel uses and unique SocketAddress which is stored in a registry. This `SocketAddress` can then be used to connect to it via the client. It will be registered as long as the server is running. Once the channel is closed, it will automatically deregister it and so the clients won't be able to access it anymore.

The behavior of connecting to a local transport server is nearly the same as with other transport implementations. One important thing to note is that you can only make use of them on the server and client side at the same time. It's not possible to use the local transport on the server side but some other on the client side. This may seem like a limitation, but once you

think about it, it makes sense. As the local transport does not bound a socket and so does not accept "real" network traffic, it's not possible to work with any other transport implementation.

4.3.4 Embedded transport

Netty also includes the embedded transport. This isn't a real transport when you compare it with the others listed previously, but it's included to complete this section. If it's not a real transport, what can it be used for?

The embedded transport allows you to interact with your different `ChannelHandler` implementation more easily. It's also easy to embed `ChannelHandler` instances in other `ChannelHandlers` and use them like a helper class.

This is often used in test cases to test a specific `ChannelHandler` implementation, but can also be used to re-use some `ChannelHandler` in your own `ChannelHandler` without event extend it. For this purpose, it comes with a concrete `Channel` implementation.

Table 4.4 Embedded channels

Channel type	Usage
`EmbeddedChannel`	Test `ChannelHandler` or embedded them in your own ChannelHandler for reuse.

Chapter 10 shows how the Embedded transport is used for testing purposes in great detail.

4.4 When to use each type of transport

Now that you've learned about all the transports in detail, you may ask when you should choose one over the other. As mentioned previously, not all transports support all core protocols. This can also limit the transports you can choose from. Table 4.6 lists the protocols that each transport supports.

Table 4.5 Transport support by protocol

Transport	TCP	UDP	SCTP*
NIO	X	X	X
OIO	X	X	X
Local			
Embedded			

* Only supported on Linux at the moment. This may change in the future..The content of the table reflects what is supported at the time of publication.

Enabling SCTP on Linux

Be aware that for SCTP, you'll need to have the user space libraries installed as well as a kernel that supports it.

For Ubuntu, use the following command:

```
# sudo apt-get install libsctp1
```

For Fedora, you use yum:

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

Please refer to the documentation of each Linux distribution for more information about how to enable SCTP.

Other than the SCTP items, there aren't any hard rules, but because of the nature of the transports there are some suggestions. It's also likely that you have to handle more concurrent connections when you implement a server than if you implement a client.

Let's take a look at the use cases that you're likely to experience.

- Low concurrent connection count

  For applications in which you expect only a low count of concurrent connections, start with the OIO transport. As with low concurrent connections, you don't need to worry too much about the limitation of having one thread allocated per connection. The resource usage will not be a problem in this scenario. You may wonder what constitutes a low-connection count; that's hard to say, but remember that with the NIO transport it's possible to serve between tens of thousands and hundreds of thousands of concurrent connections. I'd label anything under 1,000 concurrent connections as a low concurrent connection count.

  In this specific use case, you'll only gain from the low latency that the OIO transport offers, so you'll probably see a much better throughput, too.

  Anyway there are still situations in which NIO might be the better fit even with low concurrent connections count. For example if your connections are "very active" the context switching that OIO needs might be a too big impact. As always it's hard to give a hard rule here, as it all depends on your application and needs.

- High concurrent connection count

  If you expect your application to handle many concurrent connections at the same time, choose the NIO transport. These transports handle multiple concurrent connections as they don't use one thread per connection, but use a few threads and share them across the connections.

- Low latency

  If you require significantly low latency, you'll likely want to use OIO first. OIO has much less latency then NIO because of its internal design, as explain in the previous two bullet-points.

But, as always, it's a tradeoff, as you'll have to trade lower latency with higher thread count.

- Blocking code base

  If you're converting an old code base, which was heavily based on blocking networking and application design, to Netty, then you're likely to have several operations that would block the I/O thread for too long to scale efficiently with an asynchronous transport such as NIO. You could fix this by rewriting your entire stack, but this is may not be doable in the target timeframe. In that case, start with OIO and when you think you need more scale, move over to NIO.

- Communicate within the same JVM

  If you only need to communicate within the same VM and have no need to expose the service over the network, then you have the perfect use case for the local transport. This is because you'll remove all the overhead of real network operations, but still be able to reuse your Netty code base.

  This also makes it easy later to expose the service over the network if the need arises. The only thing that needs to be done in that case is to replace the transport with NIO or OIO, and maybe add an extra encoder/decoder that converts Java objects to `ByteBuf` and vice versa.

- Testing your `ChannelHandler` implementations

  If you want to write tests for your `ChannelHandler` implementations that aren't integration tests, give the embedded transport a spin. It makes it easy to test `ChannelHandlers` without the need for creating many mocks, while still conforming to the event flow that is the same in all transports. This guarantees that the `ChannelHandler` will also work once you put it to use with some real transports.

  Chapter 7 gives you more details about testing `ChannelHandlers`.

As you now know, it's important to understand what kind of use case/nature your application has, and always choose the transport that will give you the best result. Table 4.6 summarizes the common use cases.

Table 4.6 Optimal transport for an application

Application needs	Recommended transport
Low concurrent connection count	OIO
High concurrent connection count	NIO
Low latency	OIO
Blocking code base	OIO
Communication within the same JVM	Local

Testing your `ChannelHandler` implementations	Embedded

4.5 Summary

In this chapter you learned one of the fundamentals of Netty, and how it's provided to the user. You learned what a transport is and what it's used for. Also, I explained the API and gave some examples of its uses.

I highlighted the shipped transports of Netty and explained their behavior. You learned what minimum requirements the transports have, as not all transports work with the same Java version or may only work on specific operating systems.

You also learned which transport should be used for which use case, as not every transport is optimal for a specific use case.

If you're looking for more information about how you would implement your own transport and make it usable in Netty, refer to chapter 17. This will provide you with all the needed steps.

The next chapter will focus on `ByteBuf` and `MessageList`, which are "data containers" used within Netty to transfer data. You'll learn how to use it and how you can create the best performance from it.

5 Buffers

This chapter covers

- `ByteBuf`
- `ByteBufHolder`
- `ByteBufAllocator`
- Allocating and performing operations on these interfaces

Whenever you need to transmit data it must involve a buffer. Java's NIO API comes with its own `Buffer` class, but as I discussed in previous chapters, that implementation is fairly limited and not optimized. Working with the JDK's `ByteBuffer` is often cumbersome and more complex than needed. A buffer is an important component and providing the needed layer is a required task and should be part of the API.

Luckily, Netty comes with a powerful buffer implementation that's used to represent a sequence of bytes and helps you operate with either raw bytes or custom POJOs. The new buffer type, the `ByteBuf`, is effectively Netty's equivalent to the JDK's `ByteBuffer`. The `ByteBuf`'s purpose is to pass data through the Netty pipeline. It was designed from the ground up to address problems with the JDK's `ByteBuffer` and to meet the daily needs of networking application developers, making them more productive. This approach has significant advantages over using the JDK's `ByteBuffer`.

Note Throughout the rest of this book I'll refer to Netty's buffer interface and implementations as data containers to help differentiate them from the Java implementation, which I'll continue referring to as Java's buffer API.

In this chapter, you'll learn about Netty's buffer API, how it's superior to what the JDK provides out of the box, what makes it so powerful, and why it's more flexible than the JDK's buffer API. You'll gain a deeper understanding of how to access data that's exchanged in the Netty framework and how you can work with it. This chapter also builds the groundwork for later chapters, as the buffer API is used nearly everywhere in Netty.

Because the buffer passes data through Netty's `ChannelPipeline` and `ChannelHandler` implementations, the buffer is prevalent in the day-to-day development of Netty applications. `ChannelHandler` and `ChannelPipeline` will be discussed in detail in chapter 6.

5.1 Buffer API

Netty's buffer API has two interfaces:

- `ByteBuf`
- `ByteBufHolder`

Netty uses reference-counting to know when it's safe to release a `Buf` and its claimed resources. While it's useful to know that Netty uses reference counting, it's all done automatically. This allows Netty to use pooling and other tricks to speed things up and keep the memory utilization at a sane level. You aren't required to do anything to make this happen, but when developing a Netty application, you should try to process your data and release pooled resources as soon as possible.

Netty's buffer API offers several more advantages:

- You can define your own buffer type, if necessary.
- Transparent zero copy is achieved by a built-in composite buffer type.
- Capacity is expanded on demand, such as with `StringBuffer`.
- No need to call `flip()` to switch between reader/writer mode.
- Separate reader and writer index.
- Method chaining.
- Reference counting.
- Pooling.

We'll have a deeper look at some of these, including pooling, in later sections of this chapter. Let's get started with the container for bytes, which you may need first.

5.2 ByteBuf—The byte data container

Whenever you need to interact with a remote peer such as a database, the communication needs to be done in bytes. For this and other reasons an efficient, convenient, and easy-to-use data structure is required, and Netty's ByteBuf implementation meets these requirements and more, making it an ideal data container, optimized for holding and interacting with bytes.

`ByteBuf` is a data container that allows you to add/get bytes from it in an efficient way. To make it easier to operate, it uses two indices: one for reading and one for writing. This allows you to read data out of them in a sequential way and "jump" back to read it again. All you need to do is adjust the reader index and start the read operation again.

5.2.1 How it works

After something is written to the `ByteBuf`, its `writerIndex` is increased by the amount of bytes written. After you start to read bytes, its `readerIndex` is increased. You can read bytes until the `writerIndex` and `readerIndex` are at the same position. The `ByteBuf` then becomes unreadable, so the next read request triggers an `IndexOutOfBoundsException` similar to what you've seen when trying to read beyond the capacity of an array.

Calling any of the buffer's methods beginning with "read" or "write" automatically advances the reader and writer indexes or you. There are also relative operations to "set" and "get" bytes. These don't move the indexes but operate on the relative index that was given.

A `ByteBuf` may have a maximum capacity to set an upper limit to the maximum data it can hold, trying to move the writer index beyond this capacity will result in an exception. The default limit is `Integer.MAX_VALUE`.

Figure 5.2 shows how a `ByteBuf` is laid out.

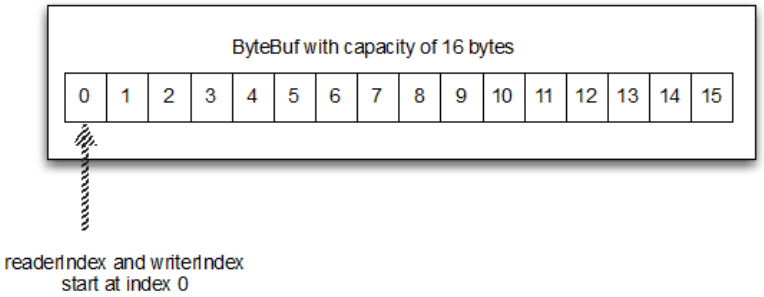


Figure 5.1 A 16-byte ByteBuf, initialized with the read and write indices set to 0

As Figure 5.1 shows, a `ByteBuf` is similar to a byte array, the most notable difference being the addition of the read and write indices which can be used to control access to the buffer's data.

You'll learn more about the operations that can be performed on a `ByteBuf` in a later section. For now, keep this in mind and let's review the different types of `ByteBuf` that you'll most likely use.

5.2.2 Different types of ByteBuf

There are three different types of `ByteBuf` you'll encounter when using Netty (there are more, but these are used internally). You may end up implementing your own, but this is out of scope here. Let's look at the provided types that you are most likely interested in.

HEAP BUFFERS

The most used type is the `ByteBuf` that stores its data in the heap space of the JVM. This is done by storing it in a backing array. This type is fast to allocate and also de-allocate when

you're not using a pool. It also offers a way to directly access the backing array, which may make it easier to interact with "legacy code".

Listing 5.1 Access backing array

```
ByteBuf heapBuf = ...;
if (heapBuf.hasArray()) {                                          #1
    byte[] array = heapBuf.array();                                #2
    int offset = heapBuf.arrayOffset() + heapBuf.position();       #3
    int length = heapBuf.readableBytes();                          #4

    YourImpl.method(array, offset, length);                        #5

}
```

#1 Check if ByteBuf is backed by array
#2 Get reference to array
#3 Calculate offset of first byte in it
#4 Get amount of readable bytes
#5 Call method using array, offset, length as parameter

Accessing the array from a "nonheap" `ByteBuf` will result in an `UnsupportedOperationException`. Because of this it's always a good idea to check if the `ByteBuf` is backed by an array with `hasArray()` as shown in listing 5.1. This pattern might be familiar if you've worked with the JDK's `ByteBuffer` before.

DIRECT BUFFERS

Another `ByteBuf` implementation is the "direct" one. Direct means that it allocates the memory directly, which is outside the "heap". You won't see its memory usage in your heap space. You must take this into account when calculating the maximum amount of memory your application will use and how to limit it, as the max heap size won't be enough. Direct buffers on the other side are optimal when it's time to transfer data over a socket. In fact, if you use a nondirect buffer, the JVM will make a copy of your buffer to a direct buffer internally before sending it over the socket.

The down side of direct buffers is that they're more expensive to allocate and de-allocate compared to heap buffers. This is one of the reasons why Netty supports pooling, which makes this problem disappear. Another possible down side can be that you're no longer able to access the data via the backing array, so you'll need to make a copy of the data if it needs to work with legacy code that requires this.

The following listing shows how you can get the data in an array and call your method even without the ability to access the backing array directly.

Listing 5.2 Access data

```
ByteBuf directBuf = ...;
if (!directBuf.hasArray()) {                                       #1
    int length = directBuf.readableBytes();                        #2
    byte[] array = new byte[length];                               #3
    directBuf.getBytes(array);                                     #4
    YourImpl.method(array, 0, array.length);                       #5
```

```
}
```

#1 Check if ByteBuf not backed by array which will be false for direct buffer
#2 Get number of readable bytes
#3 Allocate new array with length of readable bytes
#4 Read bytes into array
#5 Call method that takes array, offset, length as parameter

As you can see it's a bit more work and involves a copy operation. If you expect to access the data and need to have it in an array, you may be better off using a heap buffer.

COMPOSITE BUFFERS

The last `ByteBuf` implementation you may be confronted with is the `CompositeByteBuf`. This does exactly what its name says; it allows you to compose different `ByteBuf` instances and provides a view over them. The good thing is you can also add and remove them on-the-fly, so it's kind of like a `List`. If you've ever worked with the JDK's `ByteBuffer` you've most likely missed such a feature there. As the `CompositeByteBuf` is just a "view" over others, the `hasArray()` method will return false because it may contain several `ByteBuf` instances of both direct and nondirect types.

For example, a message could be composed of two parts: header and body. In a modularized application, the two parts could be produced by different modules and assembled later when the message is sent out. Also, you may use the same body all the time and just change the header. So it would make sense here to not allocate a new buffer every time.

This would be a perfect fit for a `CompositeByteBuf` as no memory copy will be needed and the same API could be used as with non-composite buffers.

Figure 5.2 shows how a `CompositeByteBuf` would be used to compose the header and body.

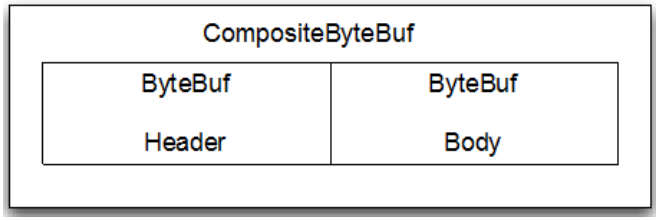


Figure 5.2 CompositeBuf that holds a header and body.

In contrast if you'd used the JDK's `ByteBuffer` this would be impossible. The only way to compose two `ByteBuffers` is to create an array that holds them or create a new `ByteBuffer` and copy the contents of both of them to the newly created one. The following listing shows these.

Listing 5.3 Compose legacy JDK `ByteBuffer`

```
// Use an array to composite them
 ByteBuffer[] message = new ByteBuffer[] { header, body };
```

```
// Use copy to merge both
ByteBuffer message2 = ByteBuffer.allocate(
                          header.remaining()+ body.remaining();
message2.put(header);
message2.put(body);
message2.flip();
```

Both approaches shown in figure 5.3 have disadvantages: having to deal with an array won't allow you to keep the API simple if you want to support both. And, of course, there's a performance cost associated with this copying; it's simply not optimal.

But let's see the `CompositeByteBuf` in action. The following listing gives an overview.

Listing 5.4 CompositeByteBuf in action

```
CompositeByteBuf compBuf = ...;
ByteBuf heapBuf = ...;
ByteBuf directBuf = ...;

compBuf.addComponent(heapBuf, directBuf);                    #1
.....
compBuf.removeComponent(0);                                  #2

for (ByteBuf buf: compBuf) {                                 #3
    System.out.println(buf.toString());
}
```

#1 Append ByteBuf instances to the composite
#2 Remove ByteBuf on index 0 (heapBuf here)
#3 Loop over all the composed ByteBuf

There are more methods in there, but I think you get the idea. The Netty API is clearly documented so as you use other methods not shown, it'll be easy to understand what they do by referring to the API docs.

Also, because of the nature of a `CompositeBytebuf`, you won't be able to access a backing array. It looks similar to what you saw for the native buffer and in the following listing.

Listing 5.5 Access data

```
CompositeBuf compBuf = ...;
if (!compBuf.hasArray()) {                                   #1
    int length = compBuf.readableBytes();                    #2
    byte[] array = new byte[length];                         #3
    compBuf.getBytes(array);                                 #4
    YourImpl.method(array, 0, array.length);                 #5

}
```

#1 Check if ByteBuf not backed by array which will be false for a composite buffer
#2 Get amount of readable bytes
#3 Allocate new array with length of readable bytes
#4 Read bytes into array
#5 Call method that takes array, offset, length as parameter

As `CompositeByteBuf` is a sub-type of `ByteBuf`, you're able to operate on the buffer as usual but with the possibility for some extra operations.

You also may welcome that Netty will optimize read and write operations on the socket whenever possible while using a `CompositeByteBuf`. This means that using gathering and scattering doesn't incur performance penalties when reading or writing to a socket or suffer from the memory leak issues in the JDK's implementation. All of this is done in the core of Netty itself so you don't need to worry about it too much, but it can't hurt to know that some optimization is done under-the-hood.

A class such as the `CompositeByteBuf` doesn't exist when using `ByteBuffer`. This is just one thing that makes the buffer API more feature-rich than the buffer API provided by the JDK as part of the `java.nio` package.

5.3 ByteBuf's byte operations

The `ByteBuf` offers many operations that allow modifying the content or just reading it. You'll learn quickly that it's much like the JDK's `ByteBuffer`, but on steroids, as it offers a much better user experience and performance.

5.3.1 Random access indexing

Like an ordinary primitive byte array, `ByteBuf` uses zero-based-indexing. This means the index of the first byte is always 0 and the index of the last byte is always capacity – 1. For example, I can iterate all the bytes of a buffer (see the following listing), regardless of its internal implementation.

Listing 5.7 Access data

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

Be aware that index pased access will not advance the `readerIndex` or `writerIndex`. You can advance it by hand by call `readerIndex(index)` and `writerIndex(index)` if needed.

5.3.2 Sequential access indexing

`ByteBuf` provides two pointer variables to support sequential read and write operations—`readerIndex` for a read operation and `writerIndex` for a write operation, respectively. Again, this is different from the JDK's `ByteBuffer` that has only one, so you need to `flip()` to switch between read and write mode. Figure 5.3 shows how a buffer is segmented into three areas by the two pointers.

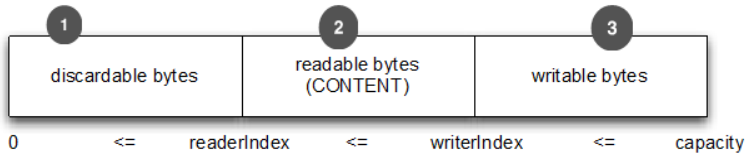


Figure 5.3 ByteBuf areas

#1 Segment that holds bytes that can be discarded as they were read before
#2 Segment that holds the actual readable content that was not read yet
#3 Segment that contains the left space of the buffer and so to which more bytes can be written

5.3.3 Discardable bytes

The discardable bytes segment contains the bytes that were already read by a read operation and so may be discarded. Initially, the size of this segment is 0, but its size increases up to the `writerIndex` as read operations are executed. This only includes "read" operations; "get" operations do not move the `readerIndex`. The read bytes can be discarded by calling `discardReadBytes()` to reclaim unused space.

Figure 5.4 shows what the segments of a `ByteBuf` look like before `discardReadBytes()` is called.

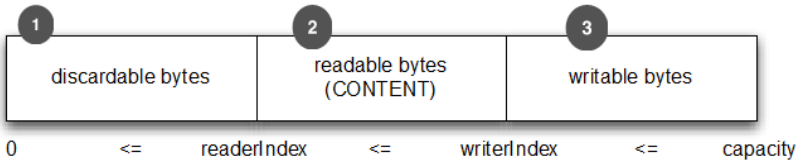


Figure 5.4 Before discardReadBytes is called.

#1 Segment that holds bytes that can be discarded as they was read before
#2 Segment that holds the actual readable content that was not read yet
#3 Segment that contains the left space of the buffer and so to which more bytes can be written

As you can see, the discardable bytes segment contains some space that is ready for reuse. This can be achieved by calling `discardReadBytes()`.

Figure 5.5 shows how the call of discardReadBytes() will affect the segments.

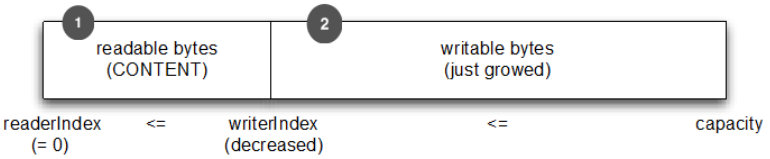


Figure 5.5 After discardReadBytes is called

#1 Segment that holds the actual readable content that was not read yet. This starts now on index 0
#3 Segment that contains the left space of the buffer and so to which more bytes can be written. This is now bigger as it growed by the space that was hold by the discardable bytes before

Note that there's no guarantee about the content of writable bytes after calling `discardReadBytes()`. The writable bytes won't be moved in most cases and could even be filled with completely different data depending on the underlying buffer implementation.

Also, you may be tempted to frequently call `discardReadBytes()` to provide the `ByteBuf` with more writable space again. Be aware that `discardReadBytes()` will most likely involve a memory copy as it needs to move the readable bytes (content) to the start of the `ByteBuf`. Such an operation isn't free and may affect performance, so only use it if you need it and will benefit from it. Thus would be for example if you need to free up memory as soon as possible.

5.3.4 Readable bytes (the actual content)

This segment is where the actual data is stored. Any operation whose name starts with read or skip will get or skip the data at the current `readerIndex` and increase it by the number of read bytes. If the argument of the read operation is also a `ByteBuf` and no destination index is specified, the specified destination buffer's `writerIndex` is increased together.

If there's not enough content left, `IndexOutOfBoundException` is raised. The default value of newly allocated, wrapped, or copied buffer's `readerIndex` is 0.

The following listing shows how to read all readable data.

Listing 5.8 Read data

```
// Iterates the readable bytes of a buffer.
ByteBuf buffer = ...;
while (buffer.readable()) {
    System.out.println(buffer.readByte());
}
```

5.3.5 Writable bytes

This segment is an undefined space which needs to be filled. Any operation whose name starts with write will write the data at the current `writerIndex` and increase it by the number of written bytes. If the argument of the write operation is also a `ByteBuf` and no source index is specified, the specified buffer's `readerIndex` is increased together.

If there's not enough writable bytes left, `IndexOutOfBoundException` is raised. The default value of newly allocated buffer's `writerIndex` is 0.

The following listing shows an example that fills the buffer with random `int` values until it runs out of space.

Listing 5.9 Write data

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

```
}
```

5.3.6 Clearing the buffer indexes

You can set both `readerIndex` and `writerIndex` to 0 by calling `clear()`. It doesn't clear the buffer's content (for example, filling with 0) but clears the two pointers. Please note that the semantics of this operation are different from the JDK's `ByteBuffer.clear()`.

Let's look at its functionality. Figure 5.6 shows a `ByteBuf` with the three different segments.

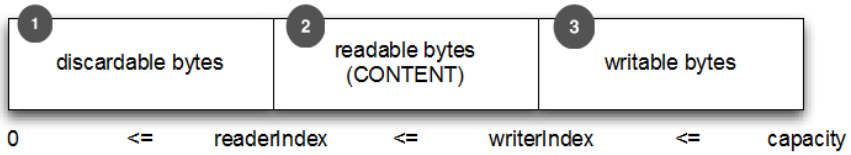


Figure 5.6 Before clear() is called.

#1 Segment that holds bytes that can be discarded as they were read before
#2 Segment that holds the actual readable content that was not read yet
#3 Segment that contains the left space of the buffer and so to which more bytes can be written

As before, it contains three segments. You'll see this change once `clear()` is called. Figure 5.7 shows the `ByteBuf` after `clear()` is used.

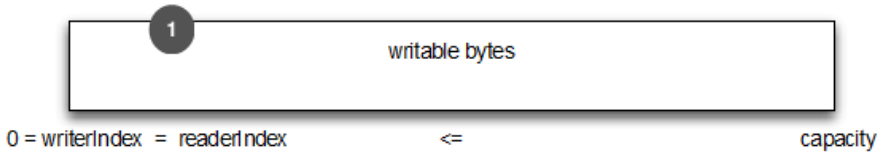


Figure 5.7 After clear() is called

#1 Segment is now as big as the capacity of the ByteBuf, so everything is writable

Compared to `discardReadBytes()`, the `clear()` operation is cheap, because it adjusts pointers and doesn't need to copy any memory.

5.3.7 Search operations

Various `indexOf()` methods help you locate an index of a value which meets a certain criteria. Complicated dynamic sequential search can be done with `ByteBufProcessor` implementations as well as simple static single-byte search.

If you're decoding variable length data such as NULL-terminated string, you'll find the `bytesBefore(byte)` method useful. Let's imagine you've written an application, which has to integrate with flash sockets, which uses NULL-terminated content. Using the `bytesBefore()` method, you can easily consume data from Flash without manually readying every byte in the

data to check for NULL bytes. Without the `ByteBufProcessor` you would need to do all this work by yourself. Also it is more efficient as it needs less "bound checks" during processing.

5.3.8 Mark and reset

As stated before, there are two marker indexes in every buffer. One is for storing `readerIndex` and the other is for storing `writerIndex`. You can always reposition one of the two indexes by calling a reset method. It works in a similar fashion to the mark and reset methods in an `InputStream` except that there are no `read limits`.

Also, you can move them to an "exact" index by calling `readerIndex(int)` or `writerIndex(int)`. Be aware that trying to set the `readerIndex` or `writerIndex` to an invalid position will cause an `IndexOutOfBoundException`.

5.3.9 Derived buffers

To create a view of an existing buffer, call `duplicate()`, `slice()`, `slice(int, int)`, `readOnly()`, or `order(ByteOrder)`. A derived buffer has an independent `readerIndex`, `writerIndex`, and marker indexes, but it shares other internal data representation the way a NIO `ByteBuffer` does. Because it shares the internal data representation, it's cheap to create and is the preferred way if, for example, you need a "slice" of a `ByteBuf` in an operation.

If a fresh copy of an existing buffer is required, use the `copy()` or `copy(int, int)` method instead. The following listing shows how to work with a slice of a `ByteBuf`.

Listing 5.10 Slice a ByteBuf

```
Charset utf8 = Charset.forName("UTF-8");
ByteBuf buf = Unpooled.copiedBuffer("Netty in Action rocks!", utf8);     #1

ByteBuf sliced = buf.slice(0, 14);                                       #2
System.out.println(sliced.toString(utf8);                                #3

buf.setByte(0, (byte) 'J');                                              #4
assert buf.get(0) == sliced.get(0);                                      #5
```

#1 Create ByteBuf which holds bytes for given string
#2 Create new slice of ByteBuf which starts at index 0 and ends at index 14
#3 Contains "Netty in Action"
#4 Update byte on index 0
#5 Won't fail as both ByteBuf share the same content and so modifications to one of them are visible on the other too

Now let's look at how to create a copy of a `ByteBuf` and how that differs from a slice. The following listing shows how to work with a copy of a `ByteBuf`.

Listing 5.11 Copying a ByteBuf

```
Charset utf8 = Charset.forName("UTF-8");
ByteBuf buf = Unpooled.copiedBuffer("Netty in Action rocks!", utf8);     #1

ByteBuf copy = buf.copy(0, 14);                                          #2
System.out.println(copy.toString(utf8);                                  #3
```

```
buf.setByte(0, (byte) 'J');                                    #4
assert buf.get(0) != copy.get(0);                              #5
```

#1 Create ByteBuf which holds bytes for given string
#2 Create copy of ByteBuf which starts at index 0 and ends at index 14
#3 Contains "Netty in Action"
#4 Update byte on index 0
#5 Won't fail as both ByteBuf does not share the same content and so modifications to one of them are not shared

The API is the same, but how a modification affects the derived `ByteBuf` is different.

Use a slice whenever possible, and use copy only as needed. Creating a copy of the `ByteBuf` is more expensive because it needs to do a memory copy.

5.3.10 Read/write operations

There are two main types of read/write operations:

- Index based get/set operations that set or get bytes on a given index.
- Read/write operations that either read bytes from the current index and increase them or write to the current index and increase it.

Let's review the relative operations first; I'll mention only the most popular for now. For a complete overview, refer to the API docs.

Table 5.1 shows the most interesting get operations that are used to access data on a given index.

Table 5.1 Relative get operations

Name	Description
`getBoolean(int)`	Return the Boolean value on the given index.
`getByte(int)` `getUnsignedByte(int)`	Return the (unsigned) byte value on the given index.
`getMedium(int)` `getUnsignedMedium(int)`	Return the (unsigned) medium value on the given index.
`getInt(int)` `getUnsignedInt(int)`	Return the (unsigned) int value on the given index.
`getLong(int)` `getUnsignedLong(int)`	Return the (unsigned) int value on the given index.
`getShort(int)` `getUnsignedShort(int)`	Return the (unsigned) int value on the given index.

`getBytes(int, ...)`	Return the (unsigned) int value on the given index.

For most of these get operations there is a similar set operation. Table 5.2 shows these.

Table 5.2 Relative set operations

Name	Description
`setBoolean(int, boolean)`	Set the Boolean value on the given index.
`setByte(int, int)`	Set byte value on the given index.
`setMedium(int, int)`	Set the medium value on the given index.
`setInt(int, int)`	Set the int value on the given index.
`setLong(int, long)`	Set the long value on the given index.
`setShort(int, int)`	Set the short value on the given index.
`setBytes(int,...)`	Transfer the bytes to the given index to/from given resources.

You may have noticed that there are no unsigned versions of these methods. This is because there isn't a notion of it when setting values. Now that you know there are relative operations, let's see them in practice, as shown in the following listing.

Listing 5.12 Relative operations on the ByteBuf

```
Charset utf8 = Charset.forName("UTF-8");
ByteBuf buf = Unpooled.copiedBuffer("Netty in Action rocks!", utf8);     #1
System.out.println((char) buf.getByte(0));                               #2

//                                                                       #3
int readerIndex = buf.readerIndex();
int writerIndex = buf.writerIndex();

buf.setByte(0, (byte) 'B');                                              #4
System.out.println((char) buf.getByte(0));                               #5

//                                                                       #6
assert readerIndex = buf.readerIndex();
assert writerIndex = buf.writerIndex();
```

#1 Create a new ByteBuf which holds the bytes for the given String
#2 Prints out the first char which is "N"
#3 Store the current readerIndex and writerIndex
#4 Update the byte on index 0 with the char 'B'
#5 Prints out the first char which is "B" now as I updated it before
#6 Check that the readerIndex and writerIndex did not change which is true as relative operations never modify the indexes.

In addition to the relative operations, there are also operations that act on the current `readerIndex` or `writerIndex`. These operations are the ones that you mostly use to read

from the `ByteBuf` as if it were a stream. The same is true for the write operations that are used to "append" to a `ByteBuf`.

Table 5.3 shows the most-used read operations.

Table 5.3 Read operations

Name	Description
`readBoolean()`	Reads the Boolean value on the current `readerIndex` and increases the `readerIndex` by 1.
`readByte()` `readUnsignedByte()`	Reads the (unsigned) byte value on the current `readerIndex` and increases the `readerIndex` by 1.
`readMedium()` `readUnsignedMedium()`	Reads the (unsigned) medium value on the current `readerIndex` and increases the `readerIndex` by 3.
`readInt()` `readUnsignedInt()`	Reads the (unsigned) int value on the current `readerIndex` and increases the `readerIndex` by 4.
`readLong()` `readUnsignedLong()`	Reads the (unsigned) int value on the current `readerIndex` and increases the `readerIndex` by 8.
`readShort()` `readUnsignedShort()`	Reads the (unsigned) int value on the current `readerIndex` and increases the `readerIndex` by 2.
`readBytes(int,int, ...)`	Reads the value on the current `readerIndex` for the given length into the given object. Also increases the `readerIndex` by the length.

There is a write method for almost every read method. Table 5.4 shows these.

Table 5.4 Write operations

Name	Description
`writeBoolean(boolean)`	Writes the Boolean value on the current `writerIndex` and increases the `writerIndex` by 1.
`writeByte(int)`	Writes the byte value on the current `writerIndex` and increases the `writerIndex` by 1.
`writeMedium(int)`	Writes the medium value on the current `writerIndex` and increases the `writerIndex` by 3.
`writeInt(int)`	Writes the int value on the current `writerIndex` and increases the `writerIndex` by 4.

`writeLong(long)`	Writes the long value on the current `writerIndex` and increases the `writerIndex` by 8.
`writeShort(int)`	Writes the short value on the current `writerIndex` and increases the `writerIndex` by 2.
`writeBytes(int,…)`	Transfers the bytes on the current `writerIndex` from given resources.

Let's see them in practice, as shown in the following listing.

Listing 5.13 Read/write operations on the ByteBuf

```
Charset utf8 = Charset.forName("UTF-8");
ByteBuf buf = Unpooled.copiedBuffer("Netty in Action rocks!", utf8);      #1
System.out.println((char) buf.readByte());                                #2

//
int readerIndex = buf.readerIndex();                                      #3
int writerIndex = buf.writerIndex();                                      #4

buf.writeByte( (byte) '?');                                               #5

//                                                                        #6
assert readerIndex = buf.readerIndex();
assert writerIndex != buf.writerIndex();
```

#1 Create ByteBuf which holds bytes for given string
#2 Prints first char "N"
#3 Store current readerIndex and writerIndex
#4 Update byte on index 0 with char "B"
#5 Prints first char "B" that I updated
#6 Check readerIndex and writerIndex didn't change

On #6, note that relative operations never modify the indexes. With a broad overview of the methods that write or read values from a `ByteBuf`, under your belt, let's take a look at a few others.

5.3.11 Other useful operations

There are other useful operations that Ihaven't mentioned yet, but they often come in handy, depending on your use case. Table 5.5 gives an overview of them and explains what they do.

Table 5.5 Other useful operations

Name	Description
`isReadable()`	Returns true if at least one byte can be read.
`isWritable()`	Returns true if at least one byte can be written.
`readableBytes()`	Returns the number of bytes that can be read.
`writablesBytes()`	Returns the number of bytes that can be written.

`capacity()`	Returns the number of bytes that the `ByteBuf` can hold. After this it will try to expand again until `maxCapacity()` is reached.
`maxCapacity()`	Returns the maximal number of bytes the `ByteBuf` can hold.
`hasArray()`	Returns true if the `ByteBuf` is backed by a byte array.
`array()`	Returns the byte array if the `ByteBuf` is backed by a byte array, otherwise throws an `UnsupportedOperationException`.

You may need to work with normal objects called POJOs. These need to be stored and retrieved later. Often, it's important to keep the order of the contained objects. For this purpose, Netty provides another data container called `MessageBuf`.

5.4 ByteBufHolder

Often, you have an object that needs to hold bytes as the actual payload and also other properties. For example, an object that represents an HTTP response is exactly like this. You have properties, such as the status code, cookies, and so on, but also the actual content/payload that's represented as bytes.

As this situation is common, Netty provides an extra "abstraction" for it called `ByteBufHolder`. The good thing is that this enables Netty to also make use of advanced features such as buffer pooling as the `ByteBuf` that holds the data can also get allocated out of a pool while still enabling Netty to release it automatically.

`ByteBufHolder`, in fact, has only a handful of methods that allows access to the data that it holds and also makes use of reference counting. Table 5.7 shows the methods that it provides (ignoring the ones that are declared in its super-type `ReferenceCounted`).

Table 5.7 ByteBufHolder operations

Name	Description
`data()`	Return the `ByteBuf` that holds the data.
`copy()`	Make a copy of the `ByteBufHolder` that does not share its data (so the data is also copied).

If you want to implement a "message object" that stores its "payload/data" in a `ByteBuf`, it's always a good idea to make use of `ByteBufHolder`.

5.4.1 Netty's buffer utility classes

One of the difficulties of using the JDK's NIO API is the amount of "boilerplate" code that's required to perform seemingly simple tasks. Even though Netty's various `Buf` implementations

are already easier to use, Netty goes even further by providing a set of utility classes which makes creating and using the various buffers even easier. In this section I'll look at three such utility classes because they'll likely be useful in your day-to-day use of Netty.

5.4.2 ByteBufAllocator—Allocate ByteBuf when needed

As mentioned before, Netty supports pooling for the various `ByteBuf` implementations. To make this possible it provides an abstraction called `ByteBufAllocator`. As the name implies it's responsible for allocating `ByteBuf` instances of the previously explained types. Whether these are pooled or not is specific to the implementation but doesn't change the way you operate with it.

Let's first look at the various operations `ByteBufAllocator` provides. Table 5.8 gives a brief overview here.

Table 5.8 ByteBufAllocator methods

Name	Description

`buffer()`
`buffer(int);`
`buffer(int, int);`
	Return a `ByteBuf` that may be of type heap or direct depend on the implementation.

`heapBuffer()`
`heapBuffer(int)`
`heapBuffer(int, int)`
	Return a `ByteBuf` of type heap.

`directBuffer()`
`directBuffer(int)`
`directBuffer(int, int)`
	Return a `ByteBuf` of type direct.

`compositeBuffer()` `compositeBuffer(int);`	
`heapCompositeBuffer ()` `heapCompositeBuffer(int);`	Return a `CompositeByteBuf` that may expand internally if needed using a heap or direct buffer.
`directCompositeBuffer ()` `directCompositeBuffer(int);`	
`ioBuffer()`	Return a `ByteBuf` that will be used for I/O operations which is reading from the socket.

As you can see, these methods take some extra arguments which allow the user to specify the initial capacity of the `ByteBuf` and the maximum capacity. You may remember that `ByteBuf` is allowed to expand as needed. This is true until the maximum capacity is reached.

Getting a reference to the `ByteBufAllocator` is easy. You can obtain it either through the channel (in theory, each channel can have another `ByteBufAllocator`) or through the `ChannelHandlerContext` that is bound to the `ChannelHandler` from which you execute your code. More on `ChannelHandler` and `ChannelHandlerContext` can be found in chapter 6.

The following listing 5.6 shows both of the ways of obtaining a byte buffer allocator.

Listing 5.6 Obtain `ByteBufAllocator` reference

```
Channel channel = ...;
ByteBufAllocator allocator = channel.alloc();                    #1
....

ChannelHandlerContext ctx = ...;
ByteBufAllocator allocator2 = ctx.alloc();                       #2
...
```

#1 Get ByteBufAllocator from a channel
#2 Get ByteBufAllocator from a ChannelHandlerContext

Netty comes with two different implementations of `ByteBufAllocator`. One implementation pools `ByteBuf` instances to minimize the allocation/de-allocation costs and keeps memory fragmentation to a minimum. How exactly these are implemented is outside the scope of this book, but let me note it's based on the "jemalloc" paper and so uses the same algorithm as many operating systems use to efficiently allocate memory.

The other implementation does not pool `ByteBuf` instances at all and returns a new instance every time. Netty uses the `PooledByteBufAllocator` (which is the pooled implementation of `ByteBufAllocator`) by default but this can be changed easily by either changing it through the `ChannelConfig` or specifying a different one when bootstrapping the server. More details can be found in chapter 9 "Bootstrap your application".

5.4.3 Unpooled—Buffer creation made easy

There may be situations where you can't access the previously explained `ByteBuf` because you don't have a reference to the `ByteBufAllocator`. For this use case Netty provides a utility class called `Unpooled`. This class contains static helper methods to create unpooled `ByteBuf` instances.

Let's have a quick peek at the provided methods. Explaining all of them would be too much for this section, so please refer to the API docs for an overview on all of them.

Table 5.9 shows the most used methods of `Unpooled`.

Table 5.9 Unpooled helper class

Name	Description
`buffer()`	
`buffer(int)`	Returns an unpooled `ByteBuf` of type `heap`.
`buffer(int, int)`	
`directBuffer()`	
`directBuffer(int)`	Returns an unpooled `ByteBuf` of type `direct`.
`directBuffer(int, int)`	
`wrappedBuffer()`	Returns a `ByteBuf`, which wraps the given data.
`copiedBuffer()`	Returns a `ByteBuf`, which copies the given data and uses it .

This `Unpooled` class also makes it easier to use the Netty's buffer API outside Netty, which you may find useful for a project that could benefit from a high-performing extensible buffer API but that does not need other parts of Netty.

5.4.4 ByteBufUtil—Small but useful

Another useful class is the `ByteBufUtil` class. This class offers static helper methods, which are helpful when operating on a `ByteBuf`. One of the main reasons to have these outside the `Unpooled` class mentioned before is that these methods are generic and aren't dependent on a `ByteBuf` being pooled or not.

Perhaps the most valuable is the `hexdump()` method which is provided as a static method like the others in this class. What it does is print the contents of the `ByteBuf` in a hex presentation. This can be useful in a variety of situations. One is to "log" the contents of the `ByteBuf` for debugging purposes. The hex value can easily be converted back to the actual byte representation. You may wonder why not print the bytes directly. The problem here is that this may result in some difficult-to-read log entries. A hex string is much more user friendly.

In addition to this the class also provides methods to check equality of `ByteBuf` implementations and others that may be of use when you start to implement your own `ByteBuf`.

5.5 Summary

In this chapter you learned about the data containers that are used inside of Netty and why these are more flexible in their usage than what you would find in the JDK.

The chapter also highlighted how you can use the different data containers and what operations are possible, and explained what operations are more "expensive" then others and what methods to use for some use cases.

In the next chapter I'll focus on `ChannelHandler`, which lets you write your own logic to process data. This `ChannelHandler` will make heavy use of the data containers described in this chapter. This will help you better understand the use cases and also show why these are important.

6 ChannelHandler

This chapter covers

- `ChannelPipeline`
- `ChannelHandlerContext`
- `ChannelHandler`
- `Inbound` versus `outbound`

Accepting connections or creating them is only one part of your application. While it's true that these tasks are important, there's another aspect that's often more complex and needs more code to write. This is the processing of incoming data and outgoing data.

Netty provides you with a powerful approach to archive exactly this. It allows the user to hook in `ChannelHandler` implementations that process the data. What makes `ChannelHandler` even more powerful is that you can "chain" them so each `ChannelHandler` implementation can fulfill small tasks. This helps you write clean and reusable implementations.

But processing data is only one thing you can do with `ChannelHandler`. You can also suppress I/O operations, which would be for example a write request (you wil see more examples later in this chapter). All of this can be done on-the-fly which makes it even more powerful.

6.1 ChannelPipeline

A `ChannelPipeline` is a list of `ChannelHandler` instances that handle or intercept inbound and outbound operations of a channel. `ChannelPipeline` offers an advanced form of the interception filter pattern, giving a user full control over how an event is handled and how the `ChannelHandlers` in the `ChannelPipeline` interact with each other.

For each new channel, a new `ChannelPipeline` is created and attached to the channel. Once attached, the coupling between the channel and the `ChannelPipeline` is permanent;

the channel cannot attach another `ChannelPipeline` to it or detach the current `ChannelPipeline` from it. All of this is handled for you; you don't need to take care of this.

Figure 6.1 describes how `ChannelHandlers` in a `ChannelPipeline` typically process I/O. An I/O operation can be handled by either a `ChannelInboundHandler` or a `ChannelOutboundHandler` and be forwarded to the closest handler by calling either one of the methods defined in the `ChannelInboundInvoker` interface for inbound I/O or by one of the methods defined in the `ChannelOutboundInvoker` interface for outbound I/O. `ChannelPipeline` extends both of them.

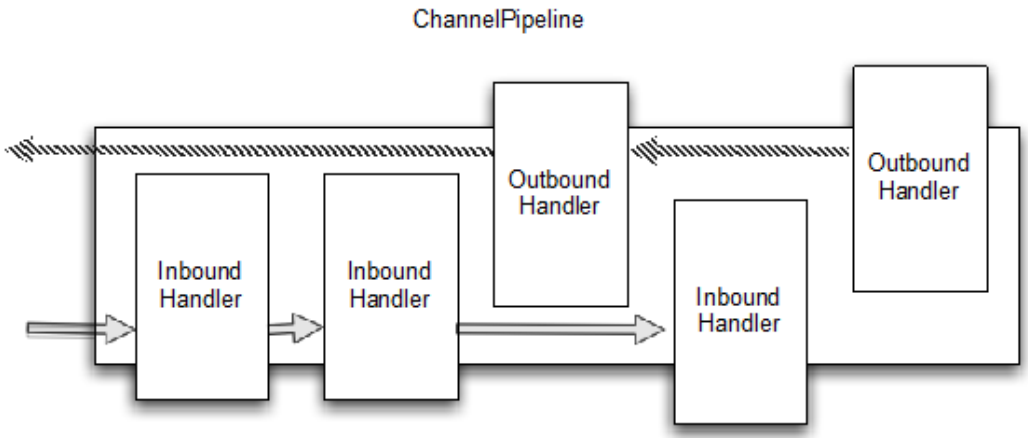


Figure 6.1 ChannelPipeline

As shown in figure 6.1 a `ChannelPipeline` is mainly a list of `ChannelHandlers`. If an inbound I/O event is triggered it's passed from the beginning to the end of the `ChannelPipeline`. For outbound I/O events it begins at the end of the `ChannelPipeline` and process to the start. The `ChannelPipeline` itself knows if a `ChannelHandler` can handle the event by checking its type. If it can't handle it, it skips the `ChannelHandler` and uses the next matching.

Modifications on the `ChannelPipeline` can be done on-the-fly, which means you can add/remove/replace `ChannelHandler` even from within another `ChannelHandler` or have it remove itself. This allows writing flexible logic, such as multiplexer, but I'll go into more detail later in this chapter.

For now, let's look at how you can modify a `ChannelPipeline`.

Table 6.1 Methods to modify a ChannelPipeline

Name	Description
`addFirst(…)`	Add a `ChannelHandler` to the `ChannelPipeline`.

`addBefore(…)`	
`addAfter(…)`	
`addLast(…)`	
`remove(…)`	Remove a `ChannelHandler` from the `ChannelPipeline`.
`replace(…)`	Rplace a `ChannelHandler` in the `ChannelPipeline` with another `ChannelHandler`.

The following listing shows how you can use these methods to modify the `ChannelPipeline`.

Listing 6.1 Modify the `ChannelPipeline`

```
ChannelPipeline pipeline = ..;
FirstHandler firstHandler = new FirstHandler();                    #1
pipeline.addLast("handler1", firstHandler);                        #2
pipeline.addFirst("handler2", new SecondHandler());                #3
pipeline.addLast("handler3", new ThirdHandler());                  #4

pipeline.remove("handler3");                                       #5
pipeline.remove(firstHandler);                                     #6

pipeline.replace("handler2", "handler4", new FourthHandler());    #7
```

#1 Create instance of FirstHandler
#2 Add FirstHandler to ChannelPipeline
#3 Add SecondHandler instance to ChannelPipeline using first position. This means it will be before the already existing FirstHandler
#4 Add ThirdHandler to ChannelPipeline on the last position
#5 Remove ThirdHandler by using name it was added with
#6 Remove FirstHandler by using reference to instance
#7 Replace SecondHandler which was added with handler2 as name by FourthHandler and add it with name handler4

As you can see, modifying the `ChannelPipeline` is easy and allows you to add, remove, replace `ChannelHandler` on demand.

ChannelHandler execution and blocking

Normally each ChannelHandler that is added to the ChannelPipeline will process the event that is passed through it in the IO-Thread, which means you MUST NOT block as otherwise you block the IO-Thread and so affect the overall handling of IO.

Sometimes it's needed to block as you may need to use legancy API's which only offers a blocking API. For example this is true for JDBC. For exactly this use-case Netty allows to pass a EventExecutorGroup to each of the ChannelPipeline.add* methods. If a custom EventExecutorGroup is passed in the event will be handled by one oft he EventExecutor contained in this EventExecutorGroup and so „moved". A default implementation which is called DefaultEventExecutorGroup comes as part of Netty.

In addition to the operations that allow you to modify the `ChannelPipeline` there are also operations that let you access the `ChannelHandler` implementations that were added as well as check if a specific `ChannelHandler` is present in the `ChannelPipeline`.

Table 6.2 Get operations on ChannelPipeline

Name	Description
`get(…)`	There are a few `get(…)` operations provided by the `ChannelPipeline`. These allow you to retrieve either the `ChannelHandler` or the `ChannelHandlerContext`, which was created and assigned to the `ChannelHandler`.
`context(…)`	Return the `ChannelHandlerContext` bound to the `ChannelHandler`.
`contains(…)`	Check if a `ChannelHandler` exists in the `ChannelPipeline` for the given name/instance.
`names()` `iterator()`	Return the names or a reference to all the added `ChannelHander` of the `ChannelPipeline`.

As `ChannelPipeline` extends `ChannelInboundInvoker` and `ChannelOutboundInvoker`, it exposes additional methods for invoking inbound and outbound operations.

Table 6.3 lists all inbound operations that are exposed, as defined in the `ChannelInboundInvoker` interface. In addition to `ChannelPipeline`, `ChannelHandlerContext` extends `ChannelInboundInvoker` and exposes those.

Table 6.3 Inbound operations on ChannelPipeline

Name	Description
`fireChannelRegistered()`	This results in having the `channelRegistered(ChannelHandlerContext)` method called of the next `ChannelInboundHandler` in the `ChannelPipeline`.

`fireChannelUnregistered()`	This results in having the `channelUnregistered(ChannelHandlerContext)` method called of the next `ChannelInboundHandler` in the `ChannelPipeline`.
`fireChannelActive()`	This results in having the `channelActive(ChannelHandlerContext)` method called of the next `ChannelInboundHandler` in the `ChannelPipeline`.
`fireChannelInactive()`	This results in having the `channelInactive(ChannelHandlerContext)` method called of the next `ChannelInboundHandler` in the `ChannelPipeline`.
`fireExceptionCaught(…)`	This results in having the `exceptionCaught(ChannelHandlerContext, Throwable)` method called of the next `ChannelHandler` in the `ChannelPipeline`.
`fireUserEventTriggered(…)`	This results in having the `userEventTriggered(ChannelHandlerContext, Object)` method call the next `ChannelInboundHandler` contained in the `ChannelPipeline`.
`fireChannelRead(….)`	This results in having the `channelRead(ChannelHandlerContext, Object msg)` method called of the next `ChannelInboundHandler` in the `ChannelPipeline`.
`fireChannelReadComplete()`	This results in having the `channelReadComplete(ChannelHandlerContext)` method called of the next `ChannelStateHandler` in the `ChannelPipeline`.

These operations are mainly useful for notifying the `ChannelInboundHandlers` in the `ChannelPipeline` and so handle various events.

But handling inbound events is only half of the story. You also need to trigger and handle outbound events which will cause some action on the underlying socket.

Table 6.4 lists all outbound operations that are exposed, as these are defined in the `ChannelOutboundInvoker` interface. In addition to `ChannelPipeline`, `ChannelHandlerContext` and `Channel` extend `ChannelOutboundInvoker`.

Table 6.4 Outbound operations on ChannelPipeline

Method name	Description
`bind(…)`	Requests to bind the `Channel` to a local address. This will call the `bind(ChannelHandlerContext, SocketAddress, ChannelPromise)` method of the next `ChannelOutboundHandler` in the `ChannelPipeline`.
`connect(…)`	Requests to connects the `Channel` to a remote address. This will call the `connect(ChannelHandlerContext, SocketAddress, ChannelPromise)` method of the next `ChannelOutboundHandler` in the `ChannelPipeline`.
`disconnect(…)`	Requests to disconnect the `Channel`. This will call the `disconnect(ChannelHandlerContext, ChannelPromise)` method of the next `ChannelOutboundHandler` in the `ChannelPipeline`.
`close(…)`	Requests to close the `Channel`. This will call the `close(ChannelHandlerContext, ChannelPromise)` method of the next `ChannelOutboundHandler` in the `ChannelPipeline`.
`deregister(…)`	Requests to deregister the `Channel` its EventLoop. This will call the `deregister(ChannelHandlerContext, ChannelPromise)` method of the next `ChannelOutboundHandler` in the `ChannelPipeline`.
`flush(…)`	Requests to flush all pending writes of the `Channel`. This will call the `flush(ChannelHandlerContext)` method of the next `ChannelOutboundHandler` in the `ChannelPipeline`.
`write()`	Requests to write the given message to the `Channel`. This will call the `write(ChannelHandlerContext, Object msg, ChannelPromise)` method of the next `ChannelOutboundHandler` in the `ChannelPipeline`. Be aware this will not write the message to the underlying Socket, but only queue it. For have it written to the actual Socket yet need to call `flush(..)` or use `writeAndFlush(…)`.
`writeAndFlush(…)`	Shortcut for calling `write(…)` and `flush(…)`.
`read()`	Requests to read more data from the `Channel`. This will call the `read(ChannelHanlderContext)` method of the next `ChannelOutboundHandler` in the `ChannelPipeline`.

6.2 ChannelHandlerContext

Each time a `ChannelHandler` is added to a `ChannelPipeline`, a new `ChannelHandlerContext` is created and assigned. The `ChannelHandlerContext` allows the `ChannelHandler` to interact with other `ChannelHandler` implementations and at the end with the underlying transport, which are part of the same `ChannelPipeline`.

The `ChannelHandlerContext` never changes for an added `ChannelHandler` so it's safe to get cached.

The `ChannelHandlerContext` does implement `ChannelInboundInvoker` and `ChannelOutboundInvoker`. It has many methods that are also present on the `Channel` or the `ChannelPipeline` itself. The difference is that if you call them on the `Channel` or `ChannelPipeline` they always flow through the complete `ChannelPipeline`. In contrast, if you call a method on the `ChannelHandlerContext`, it starts at the current position and notify the closes `ChannelHandler` in the `ChannelPipeline` that can handle the event.

6.2.1 Notify the next ChannelHandler

You can notify the closest handler in the same `ChannelPipline` by calling one of the various methods listed in `ChannelInboundInvoker` and `ChannelOutboundInvoker`. Where the notification starts depends on how you set up the notification.

Figure 6.2 shows how the `ChannelHandlerContext` belongs to the `ChannelHandler` and binds it to the `ChannelPipeline`.

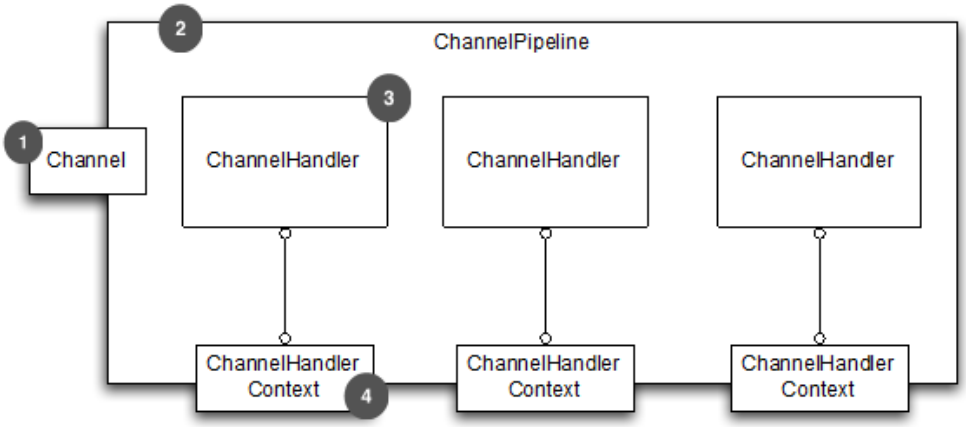


Figure 6.2 ChannelPipeline, ChannelHandlerContext and Channel

#1 Channel that ChannelPipeline is bound to
#2 ChannelPipeline bound to channel and holds added ChannelHandler instances
#3 ChannelHandler that is part of ChannelPipeline
#4 ChannelHandlerContext created while adding ChannelHandler to ChannelPipeline

Now if you'd like to have the event flow through the whole `ChannelPipeline`, there are two different ways of doing so:

- Invoke methods on the `Channel`.
- Invoke methods on the `ChannelPipeline`.

Both methods let the event flow through the whole `ChannelPipeline`. Whether it begins at the start or at the end mainly depends on the nature of the event. If it's an inbound event, it begins at the start, and if it's an outbound event it begins at the end.

The following listing shows how you can pass a write event through the `ChannelPipeline` starting at the end (as it's an outbound operation).

Listing 6.2 Events via Channel

```
ChannelHandlerContext ctx = ..;
Channel channel = ctx.channel();                                    #A
channel.write(Unpooled.copiedBuffer("Netty in Action",
        CharsetUtil.UTF_8));                                        #B
```

#A Get reference of channel that belongs to ChannelHandlerContext
#B Write buffer via channel

The message flows through the entire `ChannelPipeline`. As mentioned before, you can also do the same via the `ChannelPipeline`. The following listing shows this.

Listing 6.3 Events via `ChannelPipeline`

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

#A Get reference of ChannelPipeline that belongs to ChannelHandlerContext
#B Write buffer via ChannelPipeline

The message flows through the entire `ChannelPipeline`. Each operation (listings 6.2 and 6.3) is equal in terms of event flow. You should also notice that the `Channel` and the `ChannelPipeline` are accessible via the `ChannelHandlerContext`.

Figure 6.3 shows the flow of the event of a notification that was triggered by either the `Channel` or the `ChannelPipeline`.

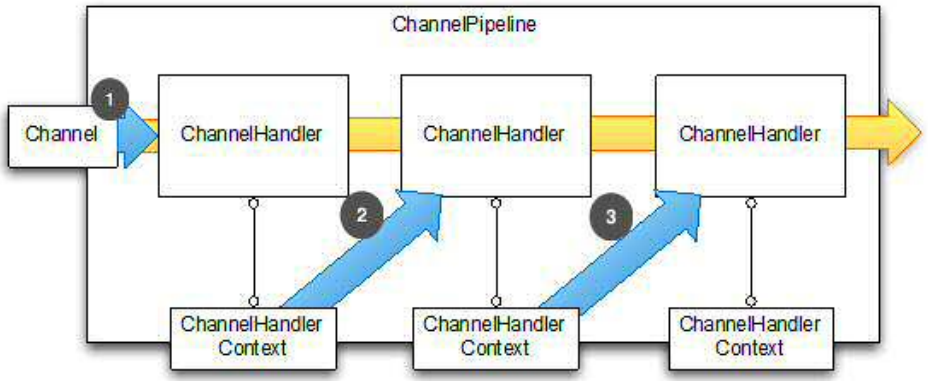


Figure 6.3 Notify via the Channel or the ChannelPipeline

#1 Event passed to first ChannelHandler in ChannelPipeline
#2 ChannelHandler passes event to next in ChannelPipeline using assigned ChannelHandlercontext
#3 ChannelHandler passes event to next in ChannelPipeline using assigned ChannelHandlercontext

There may be situations where you want to start a specific position of the `ChannelPipeline` and don't want to have it flow through the whole `ChannelPipeline`, such as:

- To save the overhead of passing the event through extra `ChannelHandlers` that are not interested in it.
- To exclude some `ChannelHandlers`.

In this case you can notify using the `ChannelHandlerContext` of the `ChannelHandler` that's your preferred starting point. Be aware that it executes the next `ChannelHandler` to the used `ChannelHandlerContext` and not the one that belongs to the used `ChannelHandlerContext`.

Listing 6.4 shows how this can be done by directly using `ChannelHandlerContext` for the operation.

Listing 6.4 Events via ChannelPipeline

```
ChannelHandlerContext ctx = ..;                                    #1
ctx.write(Unpooled.copiedBuffer("Netty in Action", CharsetUtil.UTF_8));  #2
```

#1 Get reference of ChannelHandlerContext
#2 Write buffer via ChannelHandlerContext.

The message starts to flow through the `ChannelPipeline` by the next `ChannelHandler` to the `ChanneHandlerContext` In that case the event flow would start the next `ChannelHandler` to the used `ChannelHandlerContext`.

Figure 6.4 shows the event flow.

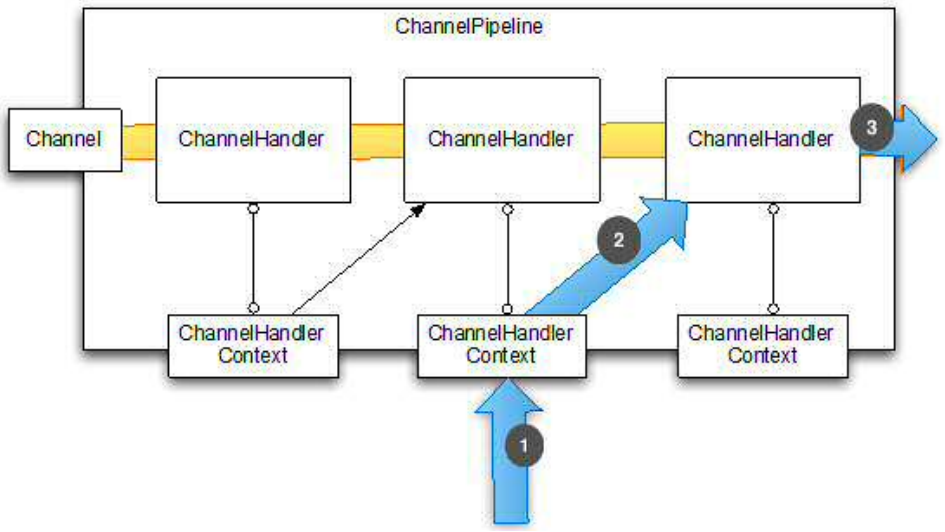


Figure 6.4 Event flow for operations triggered via the ChannelHandlerContext

#1 Event passed to specific ChannelHandler usingChannelHandlerContext
#2 Event gets passed
#3 Move out of ChannelPipeline as no ChannelHandlers remain

As you can see, it now starts at a specific `ChannelHandlerContext` and skips all `ChannelHandlers` before it. Using the `ChannelHandlerContext` for operations is a common pattern and the most used if you call operations from within a `ChannelHandler` implementation.

You can also use the `ChannelHandlerContext` from outside, because it's thread-safe.

6.2.2 Modify ChannelPipeline

You can access the `ChannelPipeline` your `ChannelHandler` belongs to by calling the `pipeline()` method. A nontrivial application could insert, remove, or replace `ChannelHandler`s in the `ChannelPipeline` dynamically in runtime.

> **NOTE** You can keep the `ChannelHandlerContext` for later use, such as triggering an event outside the handler methods, even from a different Thread.

The following listing shows how you can store the `ChannelHandlerContext` for later use and then use it even from another thread.

Listing 6.5 `ChannelHandlerContext` usage

```
public class WriteHandler extends ChannelHandlerAdapter {
```

```
    private ChannelHandlerContext ctx;

    @Override
    public void handlerAdded(ChannelHandlerContext ctx) {
        this.ctx = ctx;                                          #A
    }

    public void send(String msg) {                               #B
        ctx.write(msg);
    }
}
```

#A Store reference to ChannelHandlerContext for later use
#B Send message using previously stored ChannelHandlerContext

Please note that a `ChannelHandler` instance can be added to more than one `ChannelPipeline` if it's annotated with the `@Sharable`. This means that a single `ChannelHandler` instance can have more than one `ChannelHandlerContext`, and therefore the single instance can be invoked with a different `ChannelHandlerContext`.

If you try to add a `ChannelHandler` to more then one `ChannelPipeline` that is not annotated with `@Sharable` an exception is thrown. Be aware that once you annotate a `ChannelHandler` with `@Sharable` it must be safe to use from different threads and also be safe to use with different channels (connections) at the same time. Let's look at how it should be used. The following listing shows the correct use of the `@Sharable` annotation.

Listing 6.6 Valid usage of `@Sharable`

```
@Sharable                                                        #A
public class SharableHandler extends ChannelInboundHandlerAdapter {

    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) {
        System.out.println("Channel read message " + msg);       #B
        ctx.fireChannelRead(msg);
    }

}
```

#A Annotate with @Sharable
#B Log method call and forward to next ChannelHandler

The use of `@Sharable` is valid here, as the `ChannelHandler` doesn't use any fields to store data and so is "stateless".

There are also bad uses of `@Sharable`, as shown in the following listing.

Listing 6.7 Invalid usage of `@Sharable`

```
@Sharable                                                        #1
public class NotSharableHandler extends ChannelInboundHandlerAdapter {
    private int count;

    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) {
        count++;                                                 #2
```

```
            System.out.println("channelRead(...) called the "
                    + count + " time");                                    #3
            ctx.fireChannelRead(msg();
        }

    }
```

#1 Annotate with @Sharable
#2 Increment the count field
#3 Log method call and forward to next ChannelHandler

Why is the use of `@Sharable` wrong here? It's easy to guess once you look at the code. The problem is that we're using a field to hold the count of the method calls here. As soon as you add the same instance of the `NotSharableHandler` to the `ChannelPipeline` you get bad side effects, such as the count field being accessed and modified by different connections (and possible threads) now.

The rule of thumb is to use `@Sharable` only if you're sure that you can reuse the `ChannelHandler` on many different channels.

Why share a ChannelHandler ?

You may wonder why you want to even try to share a ChannelHandler and so annotate it with @Sharable, but there are some situation where it can make sense.

First off if you can share it you will need to create less Object which the Garbage Collector needs to recycle later. An other use case would be that you want to mantain some global statistics in the ChannelHandler which are updated by all Channels (like concurrent connection count).

6.3 The state model

Netty has a simple but powerful state model that maps perfectly to the `ChannelInboundHandler` methods. We'll look at `ChannelInboundHandler` later in the chapter. There are four different states as shown in table 6.5.

Table 6.5 The lifecycle states of a channel

State	Description
`channelUnregistered`	The channel was created, but it isn't registered to its `EventLoop`.
`channelRegistered`	The channel is registered to its `EventLoop`.
`channelActive`	The channel is active, which means it's connected to its remote peer. It's now possible to receive and send data.

`channelInactive`	The channel isn't connected to the remote peer.

The state of the `Channel` changes during its lifetime, so state changes are triggered. Typically you see four state changes during the lifetime of the `Channel`, as shown in figure 6.5.

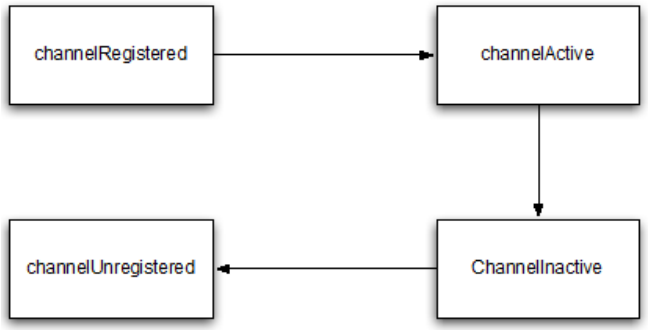


Figure 6.5 State model

In a more advanced scenario you may also see additional state changes. This is because a user is allowed to unregister the `Channel` from the `EventLoop` by hand to pause event execution of it and then later re-register it again.

In such a case you'd see more than one `channelRegistered` and `channelUnregistered` state change. You only ever have one state change for `channelActive` and `channelInactive`, because a channel can only be used for one connection lifetime; after this it needs to be recycled. If you want to reconnect, you need to create another one.

Figure 6.6 shows the state changes if a user deregisters a channel from an `EventLoop` and later reregisters it.

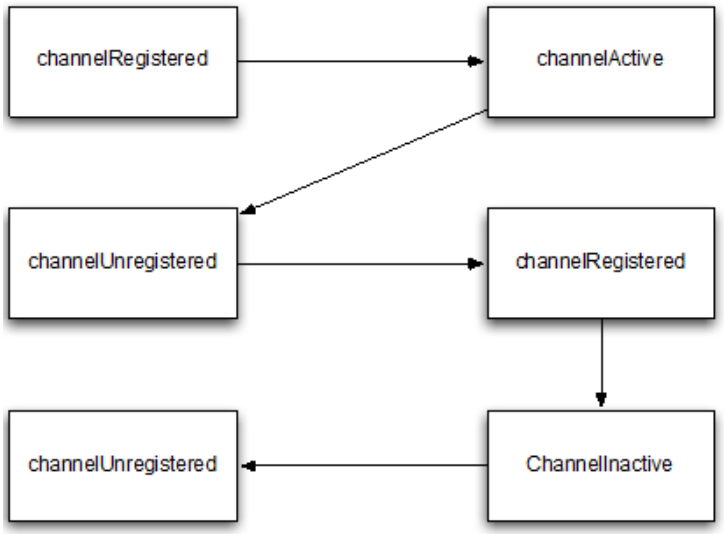


Figure 6.6 Advanced State Model

You'll learn more about the operations that can be executed on a `ByteBuf` in a later section; for now, keep this in mind while we review the different types of `ByteBuf` that you'll most likely use.

6.4 ChannelHandlers and their types

Netty supports intercept operation or reacting on state changes via `ChannelHandler`. This makes it quite easy to write your custom processing logic in a reusable way.

There are two different types of `ChannelHandlers` that Netty supports, as shown in table 6.6.

Table 6.6 ChannelHandler types

Type	Description
Inbound Handler	Processes inbound data (received data) and state changes of all kinds
Outbound Handler	Processes outbound data (to-be-sent data) and allows to incercept all kind of operations

I'll discuss each type, but let's start with the base interface for all of them.

6.4.1 ChannelHandler—the parent of all

Netty uses a well-defined type-hierarchy to represent the different handler types. The parent of all of them is the `ChannelHandler`. It provides lifecycle operations that are called after a `ChannelHandler` is added or removed from its `ChannelPipeline`.

Table 6.7 ChannelHandler methods

Type	Description
`handlerAdded(…)`	Lifecycle methods that get called during adding/removing the `ChannelHandler` from the `ChannelPipeline`.
`handlerRemoved(…)`	
`exceptionCaught(…)`	Called If an error happens during processing in the `ChannelPipeline`.

Each of the listed methods in table 6.7 takes a `ChannelHandlerContext` as a parameter when it's called. This `ChannelHandlerContext` is automatically created for each `ChannelHandler` that is added to the `ChannelPipeline`. The `ChannelHandlerContext` is bound to the `ChannelHandler`, `ChannelPipeline`, and `Channel` itself.

The `ChannelHandlerContext` allows you to safely store and retrieve values, which are local for the `Channel`. Please refer to the section about `ChannelHandlerContext` for more information about it and what you can do with it.

Netty provides a skeleton implementation for `ChannelHandler` called `ChannelHandlerAdapter`. This provides base implementations for all of its methods so you can implement (override) only the methods you're interested in. Basically what it does is forward the "event" to the next `ChannelHandler` in the `ChannelPipeline` until the end is hit.

6.4.2 Inbound handlers

Inbound handlers handle inbound events and state changes. These are the right ones if you want to react to anything that matches this criterion. In this section, we'll explore the different `ChannelHandler` subtypes that allow you to hook in the inbound logic.

CHANNELINBOUNDHANDLER

The `ChannelInboundHandler` provides methods that are called once the `Channel` state changes or data was recieved. These methods map to the channel state model explained in detail in section 6.3. Table 6.8 lists the `ChannelInboundHandler` methods.

Table 6.8 ChannelInboundHandler methods

Type	Description
`channelRegistered(…)`	Invoked once a `Channel` was registered to its `EventLoop` and is now able to handle I/O.

`channelUnregistered(…)`	Invoked once a `Channel` was deregistered from its `EventLoop` and does not handle any I/O.
`channelActive(…)`	Invoked once a `Channel` is active which means the `Channel` is connected/bound and ready.
`channelInactive(…)`	Invoked once a `Channel` change from active mode and isn't connected to its remote peer anymore.
`channelReadComplete(…)`	Invoked once a read operation on the `Channel` completed.
`channelRead(…)`	Invoked if there is something read to read from the inbound buffer.
`userEventTriggered(…)`	Invoked if the user triggered an event with some custom object.

Every one of these methods is a counterpart of a method that can be invoked on the `ChannelInboundInvoker`, which is extended by `ChannelHandlerContext` and `ChannelPipeline`.

`ChannelInboundHandler` is a subtype of `ChannelHandler` and also exposes all of its methods.

Netty provides a skeleton implementation for `ChannelInboundHandler` implementations called `ChannelInboundHandlerAdapter`. This provides base implementations for all of its methods and allows you to implement (override) only the methods you're interested in. All of these methods' implementations, by default, forward the event to the next `ChannelInboundHandler` in the `ChannelPipeline` by calling the same method on the `ChannelHandlerContext`.

Important to note is that the `ChannelInboundHandler` which handles received messages and so @Override the `channelRead(…)` method is responsible to release resources. This is especially important as Netty uses pooled resources for `ByteBuf` and so if you forgot to release the resource you will end up with a resource leak.

Listing 6.8 Handler to discard data

```
@Sharable
public class DiscardHandler extends ChannelInboundHandlerAdapter {   #1

    @Override
    public void channelRead(ChannelHandlerContext ctx,
            Object msg) {
        ReferenceCountUtil.release(msg);                              #2

    }

}
```

#1 Extend ChannelInboundHandlerAdapter

#2 Discard received message by pass it to ReferenceCountUtil.release(…)

Missing to pass the message to ReferenceCountUtil.release(…) will have the effect of creating a resource leak.

Fortunately Netty will log resources which was missed to be released with `WARN` loglevel, so it should be quite easy for you to figure out when you missed to do so.

As releasing the resources by hand can be cumbersome there is also `SimpleChannelInboundHandler`, which will take care of it for you. This way you will not need to care about it at all. The only important thing here is to remember that if you use `SimpleChannelInboundHandler`, it will release the message once it was processed and so you **MUST NOT** store a reference to it for later usage.

So how what this change the previous shown example? Listing 6.9 shows the same implementation but with the use of `SimpleChannelInboundHandler`.

Listing 6.9 Handler to discard data

```
@Sharable
public class SimpleDiscardHandler
        extends SimpleChannelInboundHandler<Object> {
#1

    @Override
    public void channelRead0(ChannelHandlerContext ctx,
            Object msg) {
        // No need to do anything special                                    #2

    }

}
```

#1 Extend SimpleChannelInboundHandler
#2 Discard received message but no need for any release of resources

If you need to be notified about other state changes you could override one of the other methods that are part of the handler.

Often you may want to decode bytes to a custom message type, and you may be tempted to implement either `ChannelInboundHandler` or extend `ChannelInboundhandlerAdapter`. But hold on a second, there's an even better solution to this problem. Those kinds of functionality are easily implemented using the codec framework, which I'll explain later in the chapter. For now, let's finish reviewing the `ChannelHandler`.

If you use `ChannelInboundHandler`, `ChannelInboundHandlerAdapter` or `SimpleChannelInboundhandler` depends on your needs. Most of the times you use `SimpleChannelInboundHandler` in the case of handling messages and `ChannelInboundHandlerAdapter` for other inbound events / state changes.

Inbound Message handling and reference counting

As you may remember from the previous chapter Netty uses reference counting to handle pooled ByteBuf's. Thus it is important to make sure the reference count is adjusted after a ByteBuf is completly processed.

Because of this it is quite important to understand how ChannelOutboundHandlerAdapter differs from SimpleChannelInboundHandler. ChannelInboundHandlerAdapter does not call „release" on the message once it was passed to channelRead(...) and so its the responsibility of the user to do so if the message was consumed. SimpleChannelInboundHandler is different here as it automatically release a message after each channelRead(...) call, and thus expect your code to either consume the message or call retain() on it if you want to use it after the method returns.

Not correctly releasing a message will produce a memory leak, lucky enough Netty will by default log a message if this happens.

CHANNELINITIALIZER

There's one slightly modified `ChannelInboundHandler` that deserves more attention: `ChannelInitializer`. It does exactly what its name states. It allows you to init the `Channel` once it's registered with its `EventLoop` and ready to process I/O.

The `ChannelInitializer` is mainly used to set up the `ChannelPipeline` for each `Channel` that is created. This is done as part of the bootstrapping (see chapter 9 for a deeper look). For now, let's remember that there's an extra `ChannelInboundHandler` for that.

6.4.3 Outbound handlers

Now that you've had a look at how `ChannelHandler` allows you to hook into inbound operations and data, it's time to look at those `ChannelHandler` implementations that allow you the same for outbound operations and data. This section shows you all of them.

CHANNELOUTBOUNDHANDLER

The `ChannelOutboundHandler` provides methods that are called when outbound operations are requested. These are the methods listed in the `ChannelOutboundInvoker` interface that's extended by `Channel`, `ChannelPipeline`, and `ChannelHandlerContext`.

What makes it powerful is the ability to implement a `ChannelOutboundHandler` and defer an operation on request. This opens some powerful and flexible ways to handle requests. For example, you could defer flush operations once nothing should be written to the remote peer and pick them up later once it's allowed again.

Table 6.9 shows all the methods that are supported.

Table 6.9 ChannelOutboundHandler methods

Type	Description

`bind(…)`	Invoked once a request to bind the `Channel` to a local address is made.
`connect(…)`	Invoked once a request to connect the `Channel` to the remote peer is made.
`disconnect(…)`	Invoked once a request to disconnect the `Channel` from the remote peer is made.
`close(…)`	Invoked once a request to close the `Channel` is made
`deregister(…)`	Invoked once a request to deregister the `Channel` from its `EventLoop` is made.
`read(…)`	Invoked once a request to read more data from the `Channel` is made.
`flush(…)`	Invoked once a request to flush queued data to the remote peer through the `Channel` is made.
`write(…)`	Invoked once a `message` should be written through the Channel to the remote peer is made.

As shown in the hierarchy, `ChannelOutboundHandler` is a subtype of `ChannelHandler` and exposes all of its methods.

Allmost all of the methods take a `ChannelPromise` as an argument that MUST be notified once the request should stop to get forwarded through the `ChannelPipeline`.

Netty provides a skeleton implementation for `ChannelOutboundHandler` implementations called `ChannelOutboundHandlerAdapter`. This provides base implementations for all of its methods and allows you to implement (override) only the methods you're interested in. All these method implementations, by default, forward the event to the next `ChannelOutboundHandler` in the `ChannelPipeline` by calling the same method on the `ChannelHandlerContext`.

Here the same is true as for the `ChannelInboundHandler`. If you handle a write operation and discard a message it's you responsible to release it probably.

Now let's look at how you could make use of this in practice. The following listing shows an implementation that discarded all written data.

Listing 6.10 Handler to discard outbound data

```
@Sharable
public class DiscardOutboundHandler
        extends ChannelOutboundHandlerAdapter {                    #1

    @Override
    public void write(ChannelHandlerContext ctx,
            Object msg, ChannelPromise promise) {
        ReferenceCountUtil.release(msg);
```

```
#2
        promise.setSuccess();                                    #3

    }

}
```

#1 Extend ChannelOutboundHandlerAdapter
#2 Release resource by using ReferenceCountUtil.release(…)
#3 Notify ChannelPromise that data handled

It's important to remember to release resources and notify the ChannelPromise. If the ChannelPromise is not notified it may lead to situations where a ChannelFutureListener is not notified about a handled message.

> **Outbound Message handling and reference counting**
>
> It's the responsiblity of the user to call ReferenceCountUtil.release(message) if the message is consumed and not passed to the next ChannelOutboundHandler in the ChannelPipleline. Once the message is passed over to the actual Transport it will be released automatically by it once the message was written or the Channel was closed.

From this examples you should be able to see how the `ChannelHandler` implementation and all of its associated features make using Netty easier and more efficient.

6.5 Summary

In this chapter you got an in-depth look into how Netty allows hooking into data processing with its `ChannelHandler` implementation. The chapter shows how `ChannelHandlers` are chained and how the `ChannelPipeline` uses them.

It highlighted the differences between inbound and outbound handlers and the differences in operating on bytes or messages of various types.

In the next chapter I'll focus on the codec abstraction of Netty, which makes writing codecs much easier than using raw `ChannelHandler` interfaces. Also, I'll take a deeper look at how you can easily test your `ChannelHandler` implementations.

7 Codec

This chapter covers

- `Codec`
- `Decoder`
- `Encoder`

In the last chapter, you learned about the various ways to hook into the processing chain and how you can intercept operations or data. While `ChannelHandler` works to implement all kinds of logic, there's still room for improvement.

To fulfill this, Netty offers a so-called codec framework that makes writing custom codecs for your protocol a piece of cake and allows for easy reuse and encapsulation.

This chapter discusses the different parts of the codec framework and how you can use it.

7.1 Codec

Whenever you write a network-based program, you'll need to implement some kind of codec. The codec defines how the raw bytes need to be parsed and converted to some kind of logic unit that represents a custom message. The same is true for converting the message back to raw bytes that can be transmitted back over the network.

Remember that everything is done by transferring bytes from one peer to another when transferring data over the network. A codec is made up of two parts:

- Decoder
- Encoder

The decoder is responsible for decoding from bytes to the message or to another sequence of bytes. The encoder does the reverse; it encodes from a message back to bytes or from bytes to another sequence of bytes.

This should make it clear that the decoder is for inbound and the encoder is for outbound data.

Message handling

As mention in Chapter 5 and Chapter 6 you need to take special care about message because of reference counting which may be used. For Decoder and Encoder the contract is quite simple. Once a message is decoded or decoded it is automatically released via ReferenceCountUtil.release(message). If you want to explicit not release the message as you may need to keep a reference for later usage etc you need to call ReferenceCountUtil.retain(message). This will make sure the reference count is incremented and so the message not released.

Most of the times you will not need this and is more for more advanced use cases.

Let's look at the various abstract base classes in Netty that allow implementing the decoder and encoder in an easy fashion.

7.2 Decoders

Netty provides a rich set of abstract base classes that help you easily write decoders. These are divided into different types:

- Decoders that decode from bytes to message
- Decoders that decode from message to message
- Decoders that decode from message to bytes

This section will give you an overview about the different abstract base classes you can use to implement your decoder and help you understand what a decoder is good for.

Before I dive into the actual abstract classes that Netty provides, let's define what the decoder's responsibility is. A decoder is responsible for decoding inbound data from one format to another one. Because a decoder handles inbound data, it's an abstract implementation of `ChannelInboundHandler`.

You may ask yourself when you'll need a decoder in practice. It is quite simple; whenever you need to transform inbound data for the next `ChannelInboundHandler` in the `ChannelPipeline`.

It's even more flexible than you may expect, as you can put as many decoders in the `ChannelPipeline` as you need, thanks to the design of the `ChannelPipeline`, which allows you to assemble your logic of reusable components.

7.2.1 ByteToMessageDecoder

Often you want to decode from bytes to messages or even from bytes to another sequence of bytes. This is such a common task that Netty ship an abstract base class that you can use to do this.

Exactly for this use case `ByteToMessageDecoder` is provided. It's an abstract base class that lets you write decoders that decode bytes into objects (POJOs) in an easy fashion.

Table 7.2 shows the most interesting methods of `ByteToMessageDecoder` and explains how these are used.

Table 7.2 Methods of ByteToMessageDecoder

Method name	Description
`decode()`	The `decode()` method is the only abstract method you need to implement. It's called with a `ByteBuf` that holds all the received bytes and a List into which decoded messages should be added. The `decode()` method is called as long as it decodes something. Default implementation delegates to `decode()`.
`decodeLast()`	This method is called once, which is when the `Channel` goes inactive. If you need special handling here you may override `decodeLast()` to implement it.

Let's imagine we have a stream of bytes written from a remote peer to us, and that it contains simple integers. We want to handle each integer separately later in the `ChannelPipeline`, so we want to read the integers from the inbound `ByteBuf` and pass each integer separately to the next `ChannelInboundHandler` in the `ChannelPipeline`.

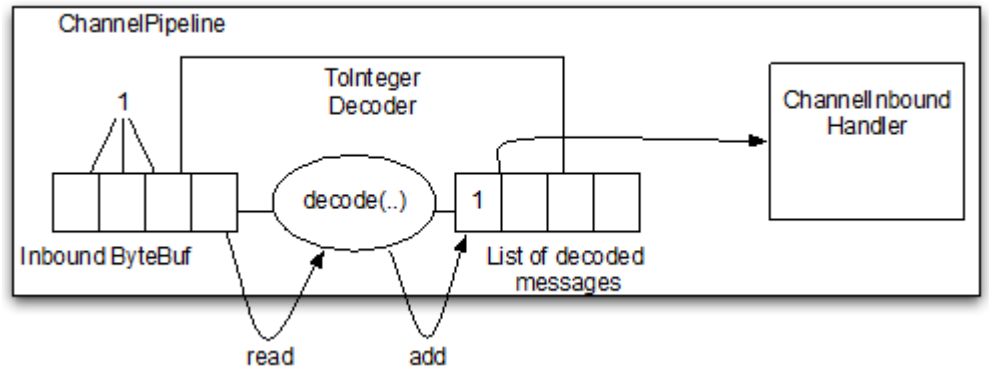


Figure 7.2 Logic of ToIntegerDecoder

You see in figure 7.2 that it will read bytes from the inbound `ByteBuf` of the `ToIntegerDecoder`, decode them, and write the decoded messages (int this case `Integer`) to the next `ChannelInboundHandler` in the `ChannelPipeline`. The figure also shows each integer will take up four bytes in the `ByteBuf`.

The following listing shows how an implementation is done for this.

Listing 7.3 ByteToMessageDecoder that decodes to Integer

```
public class ToIntegerDecoder extends ByteToMessageDecoder {           #1

    @Override
    public void decode(ChannelHandlerContext ctx, ByteBuf in,
            List<Object> out) throws Exception {
        if (in.readableBytes() >= 4) {                                  #2
            out.add(in.readInt());                                      #3
        }
    }
}
```

#1 Implementation extends ByteToMessageDecode to decode bytes to messages
#2 Check if there are at least 4 bytes readable
#3 Read integer from inbound ByteBuf, add to the List of decodec messages

Using `ByteToMessageDecoder` makes writing decoders that decode from byte to message easy as you saw in listing 7.2. But you may have noticed one little thing that is sometimes annoying. You need to check if there are enough bytes ready to read on the input `ByteBuf` before you do the actual read operation.

Wouldn't it be preferable if this was unnecessary? Yes, it sometimes is and Netty addresses this with some special decoders that allow for byte-to-message decoding. The next section will cover it.

For a more complex example, please refer to the `LineBasedFrameDecoder`. This is part of Netty itself and can be found in the `io.netty.handler.codec` package. In addition to this, there are many other implementations included, as decoding from bytes to messages is often useful.

7.2.2 ReplayingDecoder

`ReplayingDecoder` is a special abstract base class for byte-to-message decoding that would be to hard to implement if you had to check if there's enough data in the buffer all the time before calling operations on it. It does this by wrapping the input `ByteBuf` with a special implementation that checks if there's enough data ready and, if not, throws a special `Signal` that it handles internally to detect it. Once such a signal is thrown, the decode loop stops.

Because of this wrapping, `ReplayingDecoder` comes with some limitations:

- Not all operations on the `ByteBuf` are supported, and if you call an unsupported operation, it will throw an `UnreplayableOperationException`.
- `ByteBuf.readableBytes()` won't return what you expect most of the time.

If you can live with the listed limitations, you may prefer the `ReplayingDecoder` to the `ByteToMessageDecoder`. The rule of thumb is, if you can use the `ByteToMessageDecoder` without introducing too much complexity, do it. If this isn't the case, use the `ReplayingDecoder`.

Like I said before, ReplayingDecoder extends ByteToMessageDecoder, so the methods it exposes are the same. Refer to table 7.2 to see them again.

Now let's implement the `ToIntegerDecoder` with `ReplayingDecoder` in the following listing to show how it can be simplified even further.

Listing 7.4 `ReplayingDecoder` decodes to integer

```
public class ToIntegerDecoder2 extends ReplayingDecoder<Void> {        #1

    @Override
    public void decode(ChannelHandlerContext ctx, ByteBuf in,
            List<Object> out) throws Exception {
        out.add(in.readInt());                                          #2
    }
}
```

#1 Implementation extends ReplayingDecoder to decode bytes to messages
#2 Read integer from inbound ByteBuf and add it to the List of decoded messages

When reading the integer from the inbound `ByteBuf`, if not enough bytes are readable, it will throw a signal which will be cached so the `decode(…)` method will be called later, once more data is ready. Otherwise, add it to the `List`.

Compare listing 7.3 with listing 7.2 should make it easy to spot that the implementation even is even more straightforward.

Now imagine how to simplify the code if you need to implement something more complex. Again, using `ReplayingDecoder` or `ByteToMessageDecoder` is often a matter of taste. The important fact here is that Netty provides you with something you can easily use. Which one you choose is up to you.

But what do you do if you want to decode from one message to another message (for example, POJO to POJO)? This can be done using `MessageToMessageDecoder`, which is explained in the next section.

For a more complex example, please refer to the `WebSocket08FrameDecoder` or any other decoder in the `io.netty.handler.codec.http.websocketx` package.

7.2.3 MessageToMessageDecoder—Decode POJOs on the fly

If you want to decode a message to another type of message `MessageToMessageDecoder` is the easiest way to go. The semantic is quite the same as for all the other decoders I explained before.

Again I"ll have a look at the methods. Table 7.3 lists and explains them.

Table 7.3 Methods of MessageToMessageDecoder

Method name	Description
`decode()`	The `decode()` method is the only abstract method you need to implement. It's called for each inbound message of the decoder and lets you decode the message in an other message format . The decoded messages are then passed t o the next `ChannelInboundHandler` in the `ChannelPipeline`..

	Default implementation delegates to `decode()`.
`decodeLast()`	`decodeLast()` is only called one time, which is when the `Channel` goes inactive. If you need special handling here you may override `decodeLast()` to implement it.

To illustrate some uses let me give you an example. Imagine you have integers and need to convert them to a string. This should be done as part of the `ChannelPipeline` and implemented as a separate decoder to make it as flexible and reusable as possible.

Figure 7.3 shows the actual logic of the class I want to implement.

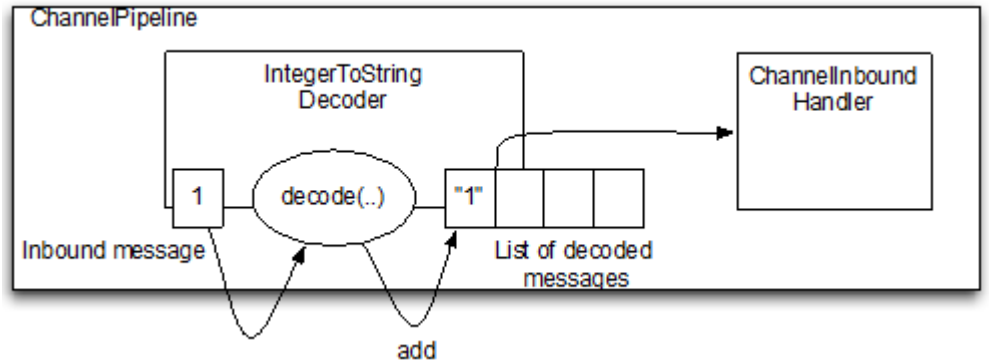


Figure 7.3 Logic of IntegerToStringDecoder

As it operates on messages and not bytes (in fact, a message can also be of type bytes). The inbound message is directly passed in the decode(…) method and decoded messages will be added to the List of decoded messages.

So the decoder will „receive" inbound messages, decode them, and add them to the List of decoded messages. Once done it will forward all decoded messages to the next `ChannelInboundHandler` in the `ChannelPipeline`.

Let's look at the concrete implementation of our needed logic in listing 7.5.

Listing 7.5 `MessageToMessageDecoder` decodes integer to string

```
public class IntegerToStringDecoder extends
        MessageToMessageDecoder<Integer> {                          #1

    @Override
    public void decode(ChannelHandlerContext ctx, Integer msg
            List<Object> out) throws Exception {
        out.add(String.valueOf(msg));                               #2
    }
```

```
}
```

#1 Implementation extends MessageToMessageDecoder
#2 Convert integer message string using String.valueOf()

In (#1), when the implementation extends the MessageToMessageDecoder to decode from one message type to another, the Type parameter (generic) is used to specify the input parameter. In our case, this is an integer type.

For a more complex example, please refer to the `HttpObjectAggregator` which can be found in the `io.netty.handler.codec.http` package.

7.2.4 Decoders summary

You should now have a good knowledge what abstract base classes Netty supports to write decoders and what decoders are good for. Decoders are only one piece of the picture. This is because most of the time you also need a way to transform outbound data. This is the job of an encoder, which is another part of the codec API.

The next sections will give you more insight into how you can write your own encoder.

7.3 Encoders

As a counterpart to the decoders Netty offers, base classes help you to write encoders in an easy way. Again, these are divided into different types similar to what you saw in section 7.2:

- Encoders that encode from message to message
- Encoders that encode from message to bytes

You may have noticed that it misses one type compared with the types listed in the decoders section. This is because it doesn't make sense to encode from byte stream to messages because if you write a `ByteBuf` to the `Channel` it will be a message of type `ByteBuf` and not a stream of bytes.

Before going deeper, let's define the responsibility of an encoder. An encoder is responsible for encoding outbound data from one format to another. As an encoder handles outbound data, it implements `ChanneOutboundHandler`.

With this in mind, it's time to look at the provided abstract classes that help implement your encoder.

7.3.1 MessageToByteEncoder

As you learned in section 7.2.2 you often want to convert from bytes to messages. You already saw how to do this for inbound data via the `ByteToMessageDecoder`. But what do you do to move back from message to bytes?

`MessageToByteEncoder` is provided to serve as an abstract base class for your encoder implementations that need to transform messages back into bytes.

Table 7.5 shows the exact method you need to implement for your encoder, which is named encode.

Table 7.5 Methods of MessageToByteEncoder

Method name	Description
`encode()`	The `encode()` method is the only abstract method you need to implement. It's called with the outbound message, which was received by this encoder and encodes it in a `ByteBuf`. The `ByteBuf` is then forwarded to the next `ChannelOutboundHandler` in the `ChannelPipeline`.

Let's see it in action to better understand its usage. I've written single short values and want to encode them into a `ByteBuf` to finally send them over the wire. `IntegerToByteEncoder` is the implementation that's used for this purpose.

Figure 7.5 shows the logic.

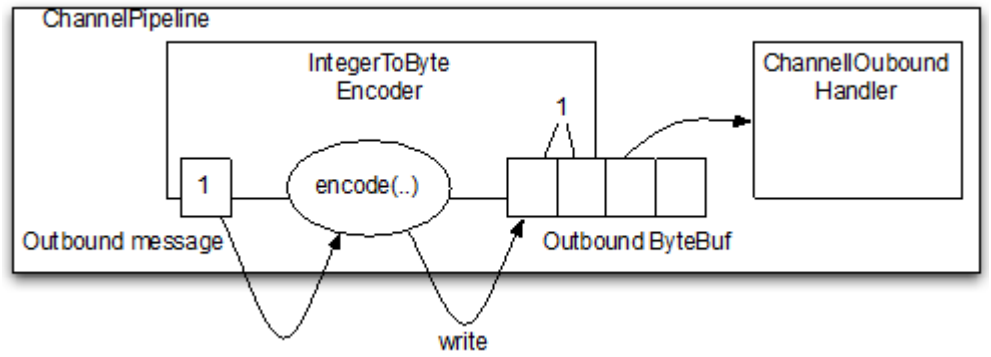


Figure 7.5 Logic of IntegerToByteEncoder

Figure 7.5 shows that it will "receive "short messages, encode, and write to a `ByteBuf`. This `ByteBuf` is then forwarded to the next `ChannelOutboundHandler` in the `ChannelPipeline`. As you can also see in figure 7.5, every short will take up 2 bytes in the `ByteBuf`.

The following listing shows how an implementation is done for this.

Listing 7.8 `IntegerToByteEncoder` encodes shorts into a `ByteBuf`

```
public class IntegerToByteEncoder extends
        MessageToByteEncoder<Short> {                                  #1

    @Override
    public void encode(ChannelHandlerContext ctx, Short msg, ByteBuf out)
            throws Exception {
        out.writeShort(msg);                                           #2
    }
```

```
}
```

#1 Implementation extends MessageToByteEncoder
#2 Write short into ByteBuf

Netty comes with several `MessageToByteEncoder` implementations which can be serve as examples to how to make use of the `MessageToByteEncoder` to create your own implementation. Please have a look at the `WebSocket08FrameEncoder` for a more real-world example. It can be found in the `io.netty.handler.codec.http.websocketx` package.

7.3.2 MessageToMessageEncoder

You've seen several decoder and encoder types. But one is still missing to complete the mix. Suppose you need a way to encode from one message to another, similar to what you did for inbound data with `MessageToMessageDecoder`.

`MessageToMessageEncoder` fills this gap. Table 7.6 shows the method you need to implement for your encoder, which is named `encode`.

Table 7.6 Methods of MessageToMessageEncoder

Name	Description
`encode()`	The `encode()` method is the only abstract method you need to implement. It's called for each message written with write(…) and encode the message to one or multiple new messages. The encoded messages are then forwarded to the next `ChannelOutboundHandler` in the `ChannelPipeline`.

Again let us look at some example. Imaging you need to encode Integer messages to `String` messages you could do this easily with `MessageToMessageEncoder`.

Figure 7.6 shows the logic.

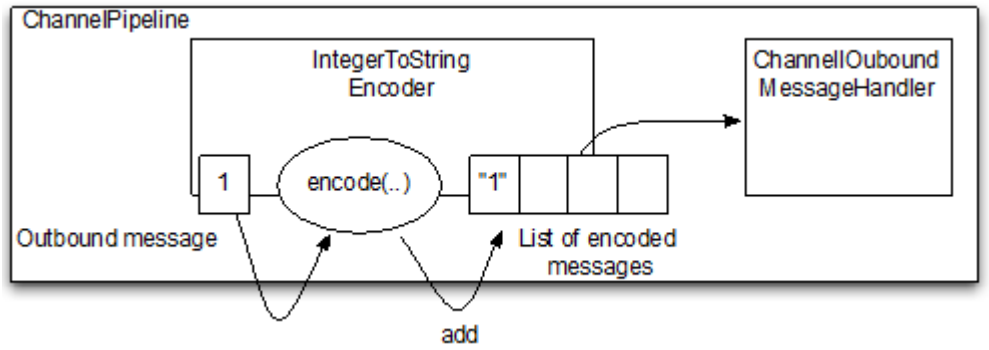


Figure 7.6 Logic of IntegerToStringEncoder

The encoder encode Integer messages and forward them to the next `ChannelOutboundHandler` in the `ChannelPipeline`.

The following listing 7.9 shows how an implementation for this is done.

Listing 7.9 `IntegerToStringEncoder` encodes integer to string

```
public class IntegerToStringEncoder extends
        MessageToMessageEncoder<Integer> {                          #1

    @Override
    public void encode(ChannelHandlerContext ctx, Integer msg
            List<Object> out) throws Exception {
        out.add(String.valueOf(msg));                               #2
    }
}
```

#1 Implementation extends MessageToMessageEncoder
#2 Convert integer to string and add to MessageBuf

Again there are more complex examples which use the `MessageToMessageEncoder`. One of those is the `ProtobufEncoderm` which can be found in the `io.netty.handler.codec.protobuf` package.

7.4 Codec

Often you want to write an implementation, which handles decoding and encoding and is encapsulated in one class. Remember, decoding is for inbound data and encoding for outbound data.

Knowing what type of encoders and decoders are provided, you won't be surprised to learn that the codecs available handle the following scenarios:

- byte-to-message decoding and encoding
- message-to-message decoding and encoding

If you're sure you'll need to have the encoder and decoder in the `ChannelPipeline` all the time and it won't work to only have one of the two, you may be better off using one of the abstract codecs.

Also, when using a codec, it's not possible to remove only the decoder or the encoder and so may leave the `ChannelPipeline` in some kind of inconsistent state. Using a codec will force you to have both the decoder and encoder in the `ChannelPipeline` or none of these.

With this in mind, let's spend the next few sections taking a deeper look at the provided abstract codec classes that make writing a codec easy.

7.4.1 ByteToByteCodec—Decoding and Encoding of bytes

If you need to decode and encode bytes to build up your codec, `ByteToByteCodec` is the right fit for you.

This is more or less a combination of `ByteToByteDecoder` and `ByteToByteEncoder`, at least in terms of functionality, as it doesn't extend both. This is because extending multiple abstract classes isn't supported in Java, at least not at the time of writing.

Because it acts like a combination of `ByteToByteDecoder` and `ByteToByteEncoder`, it also exposes the same abstract methods you need to implement. Table 7.7 lists these methods.

Table 7.7 Methods of ByteToByteCodec

Method name	Description
`decode()`	The `decode()` method is called with the inbound `ByteBuf` of the decoder and the inbound `ByteBuf` of the next `ChannelInboundByteHandler` in the `ChannelPipeline`. The `decode()` method will be called as long as bytes are consumed. Default implementation delegates to `decode()`.
`decodeLast()`	`decodeLast()` will only be called one time, which is when the `Channel` goes inactive. If you need special handling here you may override `decodeLast()` to implement it.
`encode()`	The `encode()` method is called with the outbound `ByteBuf` of the encoder and the outbound `ByteBuf` of the next `ChannelOutboundByteHandler` in the `ChannelPipeline`. The `encode()` method will be called as long as bytes are consumed.

All of this should be familiar at this point if you read the previous sections; if not, you may want to take the time to do so to get a better understanding about the responsibilities of a decoder and encoder.

So what would be a good fit for a `ByteToByteCodec`?

One is handling of compression. The decode method uncompresses and the encode method compresses. Also, it makes sense to write it in a codec, as the compressing and uncompressing are quite tight to each otherbecause the same algorithm needs to be used.

7.4.2 ByteToMessageCodec—Decoding and encoding messages and bytes

Imagine you want to have a codec that decodes bytes to some kind of message (a POJO) and encodes the message back to bytes. For this case, a `ByteToMessageCodec` is a good choice.

Again, the codec is kind of a combination. So it's like having a `ByteToMessageDecoder` and the corresponding `MessageToByteDecoder` in the `ChannelPipeline`.

Let's look at the important methods in table 7.8.

Table 7.8 Methods of ByteToByteCodec

Method name	Description
`decode()`	The `decode()` method is called with the inbound `ByteBuf` of the codec and decode them to messages. Those messages are forwarded to the next ChannelInboundHandler in the `ChannelPipeline`. The `decode()` method will be called as long as bytes are consumed . Default implementation delegates to `decode()`.
`decodeLast()`	`decodeLast()` will only be called one time, which is when the `Channel` goes inactive. If you need special handling here you may override `decodeLast()` to implement it.
`encode()`	The `encode()` method is called for each message written through the `ChannelPipeline`. The encoded messages are written in a `ByteBuf` and the `ByteBuf` is forwarded to the next `ChannelOutboundHandler` in the `ChannelPipeline`.

Let's think about a good fit for such a `ByteToMessageCodec`.

Every codec that needs to decode from bytes to some custom message and back is a good fit. For example, I've used it in the past to build a SMTP codec which reads bytes and decodes them to a custom message type called `SmtpRequest`. The encode method then took `SmtpResponse` messages and encoded them to bytes.

You see it's especially useful for requests/responses such as codecs.

7.4.3 MessageToMessageCodec—Decoding and encoding messages

Sometimes you need to write a codec that's used to convert from one message to another message type and vice versa. `MessageToMessageCodec` makes this a piece of cake.

Before going into the details, let's look at the important methods in table 7.9.

Table 7.9 Methods of MessageToMessageCodec

Method name	Description
`decode()`	The `decode()` method is called with the inbound `message` of the codec and decode them to messages. Those messages are forwarded to the next ChannelInboundHandler in the `ChannelPipeline`
`decodeLast()`	Default implementation delegates to `decode()`.

	`decodeLast()` will only be called one time, which is when the `Channel` goes inactive. If you need special handling here you may override `decodeLast()` to implement it.
`encode()`	The `encode()` method is called for each message written through the `ChannelPipeline`. The encoded messages are forwarded to the next `ChannelOutboundHandler` in the `ChannelPipeline`.

.

But where can such a codec be useful?

There are many use cases, but one of the most common is when you need to convert a message from one API to another API on the fly. This is necessary for situations in which you need to talk with a custom or old API that uses another message type.

The following listing shows this in action when converting between WebSocket frame APIs.

Listing 7.10 `WebSocketConvertHandler` converts from one API to another

```
@ChannelHandler.Sharable
public class WebSocketConvertHandler extends MessageToMessageCodec<
WebSocketFrame, WebSocketConvertHandler.WebSocketFrame> {

    public static final WebSocketConvertHandler INSTANCE = new
WebSocketConvertHandler();

    @Override
    protected void encode(ChannelHandlerContext ctx, WebSocketFrame msg
            List<Object> out) throws Exception {
        switch (msg.getType()) {
            case BINARY:
                out.add(new BinaryWebSocketFrame(msg.getData()));
                return;
            case TEXT:
                out.add(new TextWebSocketFrame(msg.getData()));
                return;
            case CLOSE:
                out.add(new CloseWebSocketFrame(true, 0, msg.getData()));
                return;
            case CONTINUATION:
                out.add(new ContinuationWebSocketFrame(msg.getData()));
                return;
            case PONG:
                out.add(new PongWebSocketFrame(msg.getData()));
                return;
            case PING:
                out.add(new PingWebSocketFrame(msg.getData()));
                return;
            default:
                throw new IllegalStateException(
                        "Unsupported websocket msg " + msg);
        }
```

```
    }

    @Override
    protected void decode(ChannelHandlerContext ctx,
            io.netty.handler.codec.http.websocketx.WebSocketFrame msg,
            List<Object> out) throws Exception {
        if (msg instanceof BinaryWebSocketFrame) {
            out.add(new WebSocketFrame(
                    WebSocketFrame.FrameType.BINARY, msg.data().copy()));
            return;
        }
        if (msg instanceof CloseWebSocketFrame) {
            out.add(new WebSocketFrame(
                    WebSocketFrame.FrameType.CLOSE, msg.data().copy()));
            return;
        }
        if (msg instanceof PingWebSocketFrame) {
            out.add(new WebSocketFrame(
                    WebSocketFrame.FrameType.PING, msg.data().copy()));
            return;
        }
        if (msg instanceof PongWebSocketFrame) {
            out.add(new WebSocketFrame(
                    WebSocketFrame.FrameType.PONG, msg.data().copy()));
            return;
        }
        if (msg instanceof TextWebSocketFrame) {
            out.add(new WebSocketFrame(
                    WebSocketFrame.FrameType.TEXT, msg.data().copy()));
            return;
        }
        if (msg instanceof ContinuationWebSocketFrame) {
            out.add(new WebSocketFrame(
                    WebSocketFrame.FrameType.CONTINUATION,
                    msg.data().copy()));
            return;
        }
        throw new IllegalStateException("Unsupported websocket msg " + msg);
    }

    public static final class WebSocketFrame {
        public enum FrameType {
            BINARY,
            CLOSE,
            PING,
            PONG,
            TEXT,
            CONTINUATION
        }

        private final FrameType type;
        private final ByteBuf data;
        public WebSocketFrame(FrameType type, ByteBuf data) {
            this.type = type;
            this.data = data;
        }
```

```
        public FrameType getType() {
            return type;
        }

        public ByteBuf getData() {
            return data;
        }
    }}
```

7.5 Other ways to compose

When you use codecs that act as a combination of a decoder and encoder, you lose the flexibility to use the decoder or encoder individually. You're forced to either have both or neither.

You may wonder if there's a way around this inflexibility problem that still allows you to have the decoder and encoder as a logic unit in the `ChannelPipeline`.

Fortunately, there is a solution for this called `CombinedChannelDuplexHandler`. Although this handler isn't part of the codec API itself, it's often used to build up a codec. .

7.5.1 CombinedChannelDuplexHandler—Combine your handlers

To show how you can use `CombinedChannelDuplexHandler` to combine a decoder and encoder, let's create those first. I'll use two simple examples to illustrate the use case.

The following listing shows a decoder, which decodes bytes to chars.

Listing 7.11 `ByteToCharDecoder` decodes bytes into chars

```
public class ByteToCharDecoder extends
        ByteToMessageDecoder {                                    #A

    @Override
    public void decode(ChannelHandlerContext ctx, ByteBuf in,
            List<Object> out) throws Exception {
        while (in.readableBytes() >= 2) {                           #B
            out.add(Character.valueOf(in.readChar()));
        }
    }
}
```

#A Extends ByteToMessageDecoder
#B Writes char into MessageBuf

Notice that the implementation extends `ByteToMessageDecoder` because it reads chars from a `ByteBuf` and adds it to the next `MessageBuf`.

Now have a look at the following listing, which represents the encoder that encodes chars back into bytes.

Listing 7.12 `CharToByteEncoder` encodes chars into bytes

```
public class CharToByteEncoder extends
        MessageToByteEncoder<Character> {                           #A

    @Override
    public void encode(ChannelHandlerContext ctx, Character msg,ByteBuf out)
```

```
            throws Exception {
        out.writeChar(msg);                                          #B
    }
}
```

#A Extends MessageToByteEncoder
#B Writes char into ByteBuf

The implementation extends `MessageToByteEncoder` because it needs to encode char messages into a `ByteBuf`.

Now that they are both present, it's time to combine them to build up a codec, as shown in the following listing.

Listing 7.13 `CharToByteEncoder` encodes chars into bytes

```
public class CharCodec extends
    CombinedChannelDuplexHandler<ByteToCharDecoder, CharToByteEncoder> { #1

    public CharCodec() {
        super(new ByteToCharDecoder(), new CharToByteEncoder());      #2
    }
}
```

#1 Handles inbound bytes and outbound messages
#2 Pass an instance of ByteToCharDecoder and CharToByteEncoder to the super constructor as it will delegate calls to them to combine them

This implementation can now be used to bundle the decoder and encoder. As you see, it's quite easy and often more flexible than using one of the *Codec classes.

What you choose to use is a matter of taste and style.

7.6 Summary

In this chapter, you learned how to use the provided codec API to write your own decoders and encoders. You also learned why using the codec API is the preferred way to write your decoders and encoders over using the plain `ChannelHandler` API.

In addition, you learned about the different abstract codec classes, which let you write a codec and let you handle decoding and encoding in one implementation.

If this isn't flexible enough, you also know how to combine a decoder and encoder to transform them to a logical unit without the need to extend any of the abstract codec classes.

In the next chapter, you'll get a brief overview about the provided `ChannelHandler` implementations and codecs that you can use out of the box to handle specific protocols and tasks.

8

Provided ChannelHandlers and codecs

This chapter covers

- Securing Netty applications with SSL/TLS
- Building Netty HTTP/HTTPS applications
- Handling idle connections and timeouts
- Decoding delimiter- and length-based protocols
- Writing big data
- Serializing data

The previous chapter showed you how to create your own codec. With this new knowledge you can now write your own codec for your application. Nevertheless, wouldn't it be nice if Netty offered some standard `ChannelHandlers` and codecs?

Netty bundles many common protocol implementations (such as HTTP) so you don’t have to reinvent the wheel. Those implementations can be reused out of the box, freeing you up to focus on more specific problems. This chapter will show you how to secure your Netty applications with SSL/TLS, how to write scalable HTTP servers, and how to use associated protocols such as WebSockets or Google's SPDY to get the best performance out of HTTP. All of these are common, and sometimes necessary, applications. This chapter will also introduce you to compression, for use when size really does matter.

8.1 Securing Netty applications with SSL/TLS

Communication of data over a network is insecure by default, as all the transmitted data may be sent in plain text or binary protocols that are easy to decode. This becomes a problem once you want to transmit data that must be private.

Encryption comes into play in this situation. TLS and SSL are well-known standards and layered on top of many protocols to make sure data remains private. For example, if you use HTTPS or SMTPS, you're using encryption. Even if you're not aware of it, you've most likely already touched base with encryption at some point.

For SSL/TLS, Java ships an abstraction called `SslContext` and `SslEngine`. In fact, the `SslContext` can be used to obtain a new `SslEngine`, which can be used for decryption and encryption. It's highly configurable to support only specific ciphers and more, but this is out of the scope of this section.

Netty extends Java's SSL engine to add functionality that makes it more suitable for Netty-based applications. It ships a `ChannelHandler` called `SslHandler` that wraps an `SslEngine` for decryption and encryption of its network traffic.

Figure 8.1 shows the data flow for the `SslHandler` implementation.

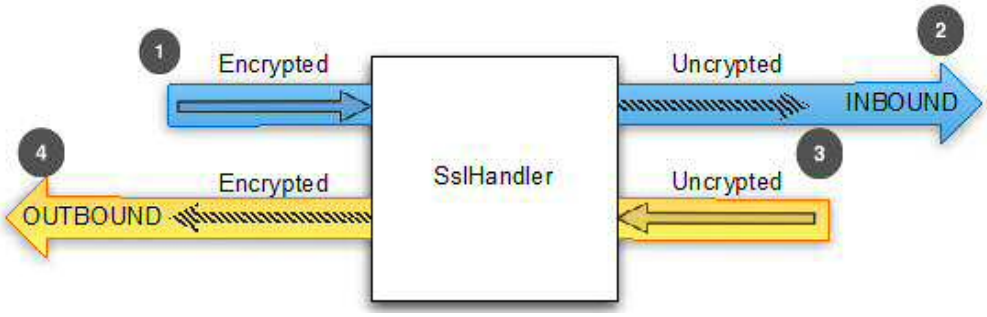


Figure 8.1 SslHandler

#1 Encrypted inbound data is intercepted by the SslHandler and gets decrypted
#2 The previous encrypted data was decrypted by the SslHandler
#3 Plain data is passed through the SslHandler
#4 The SslHandler encrypted the data and passed it outbound

Listing 8.1 shows how you can use the `SslHandler` by adding it to the `ChannelPipeline` using a `ChannelInitializer` that you typically use to set up the `ChannelPipeline` once a `Channel` is registered.

Listing 8.1 Add SSL/TLS support

```
public class SslChannelInitializer extends ChannelInitializer<Channel>{

    private final SSLContext context;
    private final boolean client;
    private final boolean startTls;

    public SslChannelInitializer(SSLContext context, boolean client,
            boolean startTls) {                                          #1
        this.context = context;
        this.client = client;
```

```
            this.startTls = startTls;
        }

        @Override
        protected void initChannel(Channel ch) throws Exception {
            SSLEngine engine = context.createSSLEngine();                    #2
            engine.setUseClientMode(client);                                 #3

            ch.pipeline().addFirst("ssl",
                    new SslHandler(engine, startTls));                       #4
        }
    }
```

#1 Use the constructor to pass the SSLContext to use and if it's a client and startTls should be used
#2 Obtain a new SslEngine from the SslContext. Use a new SslEngine for each SslHandler instance
#3 Set if the SslEngine is used in client or server mode
#4 Add the SslHandler in the pipeline as first handler

One important thing to note is that in almost all cases the `SslHandler` must be the first `ChannelHandler` in the `ChannelPipeline`. There may be some exceptions, but take this as rule of thumb. Recall in the chapter that covers `ChannelHandlers` that we said the channel pipeline is like a LIFO queue for inbound messages and a FIFO (first-in-first-out) queue for outbound messages. Adding the SSL handler first ensures that all other handlers have applied their transformations/logic to the data before it's encrypted, thus ensuring that changes from all handlers are secured on a Netty server.

The `SslHandler` also has some useful methods, as shown in table 8.1, you can use to modify its behavior or get notified once the SSL/TLS handshake is complete (during the handshake the two peers validate each other and choose an encryption cipher that both support). The SSL/TLS handshake will be executed automatically for you.

Table 8.1 Methods to modify a ChannelPipeline

Name	Description
`setHandshakeTimeout(...)` `setHandshakeTimeoutMillis(...)` `getHandshakeTimeoutMillis()`	Set and get the timeout after which the handshake will fail and the handshake `ChannelFuture` is notified.
`setCloseNotifyTimeout(...)` `setCloseNotifyTimeoutMillis(...)` `getCloseNotifyTimeoutMillis()`	Set and get the timeout after which the close notify will time out and the connection will close. This also results in having the close notify `ChannelFuture` fail.
`handshakeFuture(...)`	Returns a `ChannelFuture` that will get notified once the handshake is complete. If the handshake was done before it will return a `ChannelFuture` that contains the result of the previous handshake.
`close(...)`	Send the `close_notify` to request close and destroy

the underlying `SslEngine`.

8.2 Building Netty HTTP/HTTPS applications

HTTP/HTTPS is one of the most used protocols these days, and with the success of smartphones it gets more attention with each passing day. Almost every company has a homepage these days that you can access via HTTP or HTTPS, but this isn't the only use for it. Many services expose an API via HTTP/HTTPS for easy use in a platform-independent manner.

Fortunately, Netty comes with handlers that allow you to use HTTP without requiring you to write your own codecs and so on.

8.2.1 Netty's HTTP decoder, encoder, and codec

HTTP uses a Request–Response pattern, which means the client sends an HTTP request to the server and the server sends an HTTP response back. Netty makes it easy to work with HTTP by providing various encoders and decoders for handling HTTP protocols. Figures 8.2 and 8.3 shows the commands that reproduce and handle the HTTP-specific messages for requests and responses, respectively.

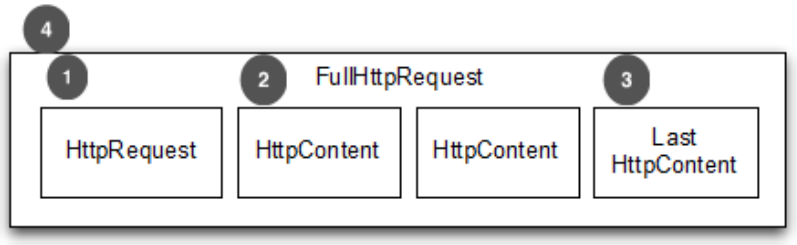


Figure 8.2 HTTP request parts

#1 First part of the HTTP-Request that contains headers and so on
#2 One more HttpContent part that contains chunks of data
#3 Special HttpContent subtype that marks the end of the HTTP request and may also contain trailing headers
#4 A full HTTP request with everything included

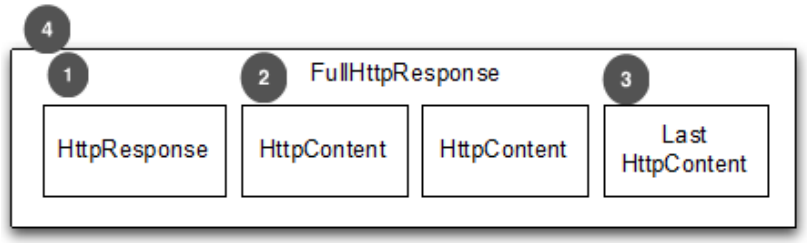


Figure 8.3 HTTP response parts

#1 First part of the HTTP response that contains headers and so on
#2 One more HttpContent part that contains chunks of data
#3 Special HttpContent subtype that marks the end of the HTTP response and may also contain trailing headers
#4 A full HTTP response with everything included

As shown in figures 8.2 and 8.3, respectively, an HTTP request/response may consist of more than one message. Its end is always marked with the `LastHttpContent` message. The `FullHttpRequest` and `FullHttpResponse` commands are a special subtype that represent a complete request and response. All types of HTTP messages (`FullHttpRequest`, `LastHttpContent`, and those shown in listing 8.2) implement the `HttpObject` interface as their parent.

Table 8.2 gives an overview of the HTTP decoders and encoders that handle and produce the messages.

Table 8.2 HTTP decoder and encoder

Name	Description
`HttpRequestEncoder`	Encodes `HttpRequest` and `HttpContent` messages to bytes.
`HttpResponseEncoder`	Encodes `HttpResponse` and `HttpContent` messages to bytes.
`HttpRequestDecoder`	Decodes bytes into `HttpRequest` and `HttpContent` messages.
`HttpResponseDecoder`	Decodes bytes into `HttpResponse` and `HttpContent` messages.

Listing 8.2 Add support for HTTP

```
public class HttpDecoderEncoderInitializer
        extends ChannelInitializer<Channel> {

    private final boolean client;

    public HttpDecoderEncoderInitializer(boolean client) {
        this.client = client;
    }

    @Override
    protected void initChannel(Channel ch) throws Exception {
        ChannelPipeline pipeline = ch.pipeline();
        if (client) {
            pipeline.addLast("decoder", new HttpResponseDecoder());      #1
            pipeline.addLast("encoder", new HttpRequestEncoder());       #2
        } else {
            pipeline.addLast("decoder", new HttpRequestDecoder());       #3
            pipeline.addLast("encoder", new HttpResponseEncoder());      #4
```

```
            }
        }
    }
```

#1 Add HttpResponseDecoder as decoder as a client will receive responses from the server
#2 Add HttpRequestEncoder as encoder as the client will send requests to the server
#3 Add HttpRequestDecoder as decoder as the server will receive request from the client
#4 Add HttpResponseEncoder as encoder as the server will send responses to the client

If you need to have a decoder and encoder in the `ChannelPipeline`, there's also a codec to simplify things for the client and server. So instead of adding the decoder and encoder, use `HttpClientCodec` or `HttpServerCodec`.

After you have the decoder, encoder, or codec in the `ChannelPipeline`, you'll be able to operate on the different `HttpObject` messages. But as an HTTP request and HTTP response can be made out of many "messages," you'll need to handle the different parts and may have to aggregate them. This can be cumbersome. To solve this problem, Netty provides an aggregator, which will merge message parts into `FullHttpRequest` and `FullHttpResponse` messages, so you don't need to worry about receiving only "fragments." The next section will explain message aggregation in more detail.

8.2.2 HTTP message aggregation

As explained in the previous section, when dealing with HTTP you may receive HTTP messages in fragments, because otherwise Netty would need to buffer the whole message until it's received.

Situations exist where you want to handle only "full" HTTP messages and be okay with some memory overhead. For this purpose Netty bundles the `HttpObjectAggregator`.

With the `HttpObjectAggregator` Netty will aggregate HTTP message parts for you and only forward `FullHttpResponse` and `FullHttpRequest` to the next inbound `ChannelHandler` in the `ChannelPipeline`. This eliminates the worry about fragmentation and ensures you act only on "complete" messages.

Automatic aggregation is as easy as adding another `ChannelHandler` in the `ChannelPipeline`. Listing 8.3 shows how to enable the aggregation.

Listing 8.3 Automatically aggregate HTTP message fragments

```
public class HttpAggregatorInitializer extends ChannelInitializer<Channel> {

    private final boolean client;

    public HttpAggregatorInitializer(boolean client) {
        this.client = client;
    }

    @Override
    protected void initChannel(Channel ch) throws Exception {
        ChannelPipeline pipeline = ch.pipeline();
        if (client) {
            pipeline.addLast("codec", new HttpClientCodec());          #1
        } else {
            pipeline.addLast("codec", new HttpServerCodec());          #2
```

```
            }
            pipeline.addLast("aggegator",
                new HttpObjectAggregator(512 * 1024));                    #3
        }
    }
```

#1 Add HttpClientCodec as we are in client mode
#2 Add HttpServerCodec as we are in server mode
#3 Add HttpObjectAggregator to the ChannelPipeline, using a max message size of 512kb. After the message is getting bigger a TooLongFrameException is thrown.

As you can see, it's easy to let Netty automatically aggregate the message parts for you. Be aware that to guard your server against DoS attacks you need to choose a sane limit for the maximum message size. How big the maximum message size should be depends on your use case, concurrent requests/responses, and, of course, the amount of usable memory available.

8.2.3 HTTP compression

When using HTTP it's often advisable to use compression to minimize the data you need to transfer over the wire. The performance gain from this isn't free, however, as compression puts more load on the CPU. Although today's increased computing capabilities mean that this isn't as much of an issue as a few years ago, using compression is a good idea most of the time.

Netty comes out of the box with "gzip" and "deflate" support provided by two `ChannelHandler` implementations: one for compression and one for decompression. Compression and decompression are shown in listing 8.4.

Listing 8.4 Automatically compress HTTP messages

```
public class HttpAggregatorInitializer extends ChannelInitializer<Channel> {

    private final boolean client;

    public HttpAggregatorInitializer(boolean client) {
        this.client = client;
    }

    @Override
    protected void initChannel(Channel ch) throws Exception {
        ChannelPipeline pipeline = ch.pipeline();
        if (client) {
            pipeline.addLast("codec", new HttpClientCodec());             #1
                pipeline.addLast("decompressor",
                        new HttpContentDecompressor());
#2
        } else {
            pipeline.addLast("codec", new HttpServerCodec());             #3
                pipeline.addLast("decompressor",
                        new HttpContentDecompressor());
#4
        }
}
}
```

#1 Add HttpClientCodec as we are in client mode
#2 Add HttpContentDecompressor as the server may send us compressed content

#3 Add HttpServerCodec as we are in server mode
#4 Add HttpContentCompressor as the client may support compression and if so we want to compress it

8.2.4 Using HTTPS

As mentioned before in this chapter, you may want to protect your network traffic using encryption. You can do this via HTTPS, which is a piece of cake thanks to the "stackable" `ChannelHandlers` that come with Netty.

All you need to do is to add the `SslHandler` to the mix, as shown in listing 8.5.

Listing 8.5 Using HTTPS

```
public class HttpsCodecInitializer extends ChannelInitializer<Channel> {

    private final SSLContext context;
    private final boolean client;

    public HttpsCodecInitializer(SSLContext context, boolean client) {
        this.context = context;
        this.client = client;
    }

    @Override
    protected void initChannel(Channel ch) throws Exception {
        ChannelPipeline pipeline = ch.pipeline();
        SSLEngine engine = context.createSSLEngine();
        engine.setUseClientMode(client);
        pipeline.addFirst("ssl", new SslHandler(engine));               #1

        if (client) {
            pipeline.addLast("codec", new HttpClientCodec());           #2
        } else {
            pipeline.addLast("codec", new HttpServerCodec());           #3
        }
    }
}
```

#1 Add SslHandler to use HTTPS
#2 Add HttpClientCodec as we are in client mode
#3 Add HttpServerCodec as we are in server mode

Listing 8.5 demonstrates how Netty's pipeline enables flexible applications. This section is a good example of how handlers enable a powerful approach to network programming. HTTP is one of the most widely used protocols on the World Wide Web and Netty comes equipped with all of the necessary tools to help you develop HTTP-based applications quickly and easily. In the next section we'll look at one of the newest extensions to the HTTP protocol: WebSockets.

8.2.5 Using WebSockets

HTTP is nice, but what do you do if you need to publish information in real time? Previously, you might have used a workaround such as long-polling, but that kind of solution isn't optimal and can be kind of slow.

These reasons are exactly why the WebSockets specification and implementations were created. WebSockets allow exchanging data in both directions (from client to server and from server to client) without the need of a request-response pattern. While in the early days of WebSockets it was only possible to send text data via WebSockets, this is no longer true. Now you can also send binary data, which makes it possible to build everything you want on top of WebSockets.

We won't go into too much detail about how WebSockets works as it's out of scope for this book, but the general idea of what happens in a WebSocket communication is shown in figure 8.4.

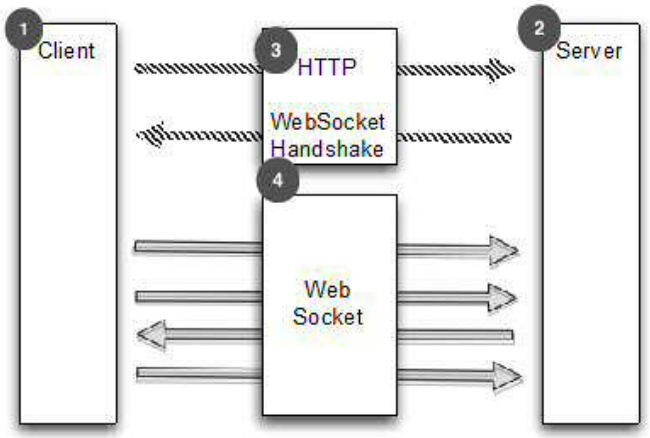


Figure 8.4 WebSocket

#1 Client that communicates with server
#2 Server that communicates with client
#3 Client issues WebSocket handshake via HTTP(s) and waits for completion
#4 Connection upgraded to WebSockets

As you saw in figure 8.4 the connection starts as HTTP and then "upgrades" to WebSockets. Adding support for WebSockets in your application is easy because Netty supports it via its `ChannelHandler` implementations.

When using WebSockets there will be different message types you need to handle. Table 8.3 shows them.

Table 8.3 WebSocketFrame types

Name	Description
`BinaryWebSocketFrame`	`WebSocketFrame` that contains binary data.

`TextWebSocketFrame`	`WebSocketFrame` that contains text data.
`ContinuationWebSocketFrame`	`WebSocketFrame` that contains text or binary data that belongs to a previous `BinaryWebSocketFrame` or `TextWebSocketFrame`.
`CloseWebSocketFrame`	`WebSocketFrame` that represents a CLOSE request and contains close status code and a phrase.
`PingWebSocketFrame`	`WebSocketFrame` which request the send of a `PongWebSocketFrame`.
`PongWebSocketFrame`	`WebSocketFrame` which is sent as response to a `PingWebSocketFrame`.

To keep it short we'll only have a look at what to use for a WebSocket server. More information for a client can be found in the examples that ship with Netty source code.

Netty offers many ways to use WebSockets, but the easiest one, which works for most users, is using the `WebSocketServerProtocolHandler` when writing a WebSockets server, as shown in listing 8.6. This handles the handshake and also the `CloseWebSocketFrame`, `PingWebSocketFrame`, and `PongWebSocketFrame` for you.

Listing 8.6 Support WebSocket on the server

```
public class WebSocketServerInitializer extends ChannelInitializer<Channel>{
    @Override
    protected void initChannel(Channel ch) throws Exception {
        ch.pipeline().addLast(
                new HttpServerCodec(),
                new HttpObjectAggregator(65536),                          #1
                new WebSocketServerProtocolHandler("/websocket"),         #2
                new TextFrameHandler(),                                   #3
                new BinaryFrameHandler(),                                 #4
                new ContinuationFrameHandler());                          #5
    }

    public static final class TextFrameHandler extends
            SimpleChannelInboundHandler<TextWebSocketFrame> {
        @Override
        public void channelRead0(ChannelHandlerContext ctx,
                TextWebSocketFrame msg) throws Exception {
            // Handle text frame
        }
    }

    public static final class BinaryFrameHandler extends
            SimpleChannelInboundHandler<BinaryWebSocketFrame> {
        @Override
        public void channelRead0(ChannelHandlerContext ctx,
                BinaryWebSocketFrame msg) throws Exception {
            // Handle binary frame
        }
```

```
    }

    public static final class ContinuationFrameHandler extends
            SimpleChannelInboundHandler<ContinuationWebSocketFrame> {
        @Override
        public void channelRead0(ChannelHandlerContext ctx,
                ContinuationWebSocketFrame msg) throws Exception {
            // Handle continuation frame
        }
    }
}
```

#1 Add HttpObjectAggregator as we need aggregated HttpRequests for the handshake
#2 Add WebSocketServerProtocolHandler will handle the upgrade if a request is send to /websocket and also handle the Ping, Pong and Close frames after the upgrade
#3 TextFrameHandler will handle TextWebSocketFrames
#4 BinaryFrameHandler will handle BinaryWebSocketFrames
#5 ContinuationFrameHandler will handle ContinuationWebSocketFrames

If you want to support Secure-WebSocket, it's as easy as adding the `SslHandler` as the first `ChannelHandler` in the `ChannelPipeline`.

8.2.6 SPDY

SPDY (SPeeDY) is one of the new movements when it comes to HTTP. Google invented SPDY, but you can find some of its "ideas" in the upcoming HTTP 2.0.

The idea of SPDY is to make transfer of content much faster. This is done by:

- GZIPing everything
- Encrypting everything
- Allowing multiple transfers per connection
- Providing support for different transfer priorities

At the time of writing there are three versions of SPDY in the wild. Those versions are based on:

- Draft 1 – Initial version
- Draft 2 – Contains fixed and new features such as server-push
- Draft 3 – Contains fixes and features such as flow control

Many web browsers support SPDY at the time of writing, including Google Chrome, Firefox, and Opera.

Netty comes with support for Draft 2 and Draft 3, which are the most used at the moment and should allow you to support most of your users.

We won't cover SPDY's use in detail here as there's a full chapter on it later in this book.

8.3 Handling idle connections and timeouts

You may find times when you want to handle idle connections and timeouts. Typically you'd send a message, also called a "heartbeat," to the remote peer if it idles too long in order to detect if it's still alive. Another approach is to disconnect the remote peer if it idles too long.

Dealing with idle connections is such a core part of many applications that Netty ships a solution for it. Three different `ChannelHandlers` handle idle and timeout, as shown in table 8.4.

Table 8.4 ChannelHandlers for handling idle and timeouts

Name	Description
`IdleStateHandler`	`IdleStateHandler` fires an `IdleStateEvent` if the connection idles too long. You can then act on the `IdleStateEvent`.
`ReadTimeoutHandler`	`ReadTimeoutHandler` throws a `ReadTimeoutException` and closes the `Channel` when there is no inbound data received for the timeout.
`WriteTimeoutHandler`	`WriteTimeoutHandler` throws a `WriteTimeoutException` and closes the `Channel` when there is no inbound data received for the timeout.

The most used in practice is the `IdleStateHandler`, so let's focus on it next.

Listing 8.9 shows how you can use the IdleStateHandler to get notified if you haven't received or sent data for 60 seconds. If this is the case, a heartbeat will be written to the remote peer, and if this fails the connection is closed.

Listing 8.9 Send heartbeats on idle

```
public class IdleStateHandlerInitializer extends ChannelInitializer<Channel>
{

    @Override
    protected void initChannel(Channel ch) throws Exception {
        ChannelPipeline pipeline = ch.pipeline();
        pipeline.addLast(
                new IdleStateHandler(0, 0, 60, TimeUnit.SECONDS));      #1
            pipeline.addLast(new HeartbeatHandler());
    }

    public static final class HeartbeatHandler extends
ChannelStateHandlerAdapter {
        private static final ByteBuf HEARTBEAT_SEQUENCE =
                Unpooled.unreleasableBuffer(Unpooled.copiedBuffer(
                    "HEARTBEAT", CharsetUtil.ISO_8859_1));      #2

        @Override
        public void userEventTriggered(ChannelHandlerContext ctx,
                Object evt) throws Exception {
            if (evt instanceof IdleStateEvent) {
                 ctx.writeAndFlush(HEARTBEAT_SEQUENCE.duplicate())
                         .addListener(
                              ChannelFutureListener.CLOSE_ON_FAILURE);  #3
```

```
            } else {
                super.userEventTriggered(ctx, evt);                    #4
            }
        }
    }
}
```

#1 Add IdleStateHandler which will fire an IdleStateEvent if the connection has not received or send data for 60 seconds
#2 The heartbeat to send to the remote peer
#3 Send the heartbeat and close the connection if the send operation fails
#4 Not of type IdleStateEvent pass it to the next handler in the ChannelPipeline

8.4 Decoding delimiter- and length-based protocols

As you work with Netty, you'll come across delimiter-based and length-based protocols that need to be decoded. This section explains the implementations that come with Netty for the purpose of decoding these protocols.

8.4.1 Delimiter-based protocols

Often you need to handle delimiter-based protocols or build on top of them. Some examples of delimiter-based protocols are SMTP, POP3, IMAP, and Telnet, to name a few. For those, Netty ships with special handlers that make it easy to extract frames delimited by some sequence.

Table 8.5 ChannelHandler for handling idle and timeouts

Name	Description
`DelimiterBasedFameDecoder`	`Decoder` that extracts frames for a given delimiter.
`LineBasedFrameDecoder`	`Decoder` that extracts frames for \r \n delimiter. This is faster than `DelimiterBasedFrameDecoder`.

So how does it work? Let's have a look at figure 8.5, which shows how frames are handled when delimited by a "\r\n" sequence.

Figure 8.5 Delimiter terminated frame handling

#1 Stream of bytes
#2 First frame which was extracted out of the stream
#3 Second frame which was extracted out of the stream

Listing 8.10 shows how you can use the `LineBasedFrameDecoder` to extract the "\r\n" delimited frames.

Listing 8.10 Handling \r\n delimited frames

```
public class LineBasedHandlerInitializer extends ChannelInitializer<Channel>
{

    @Override
    protected void initChannel(Channel ch) throws Exception {
        ChannelPipeline pipeline = ch.pipeline();
        pipeline.addLast(new LineBasedFrameDecoder(65 * 1024));          #1
        pipeline.addLast(new FrameHandler());                            #2
    }

    public static final class FrameHandler
            extends SimpleChannelInboundHandler<ByteBuf> {
        @Override
        public void channelRead0(ChannelHandlerContext ctx,
                ByteBuf msg) throws Exception {                          #3
            // Do something with the frame
        }
    }
}
```

#1 Add LineBasedFrameDecoder which will extract the frames and forward to the next handler in the pipeline
#2 Add FrameHandler that will receive the frames
#3 Do something with the frame

If your frames are delimited by something other than line breaks, you can use the `DelimiterBasedFrameDecoder` in a similar fashion. You only need to pass the delimiter to the constructor.

Those decoders are also useful if you want to implement your own delimiter-based protocol. Imagine you have a protocol that handles only commands. Those commands are formed out of a name and arguments. The name and the arguments are separated by a whitespace.

Writing a decoder for this is a piece of cake if you extend the `LineBasedFrameDecoder`. Listing 8.11 shows how to do it.

Listing 8.11 Decoder for the command and the handler

```
public class CmdHandlerInitializer extends ChannelInitializer<Channel> {

    @Override
    protected void initChannel(Channel ch) throws Exception {
        ChannelPipeline pipeline = ch.pipeline();
        pipeline.addLast(new CmdDecoder(65 * 1024));                     #1
        pipeline.addLast(new CmdHandler());                              #2
    }

    public static final class Cmd {                                      #3
        private final ByteBuf name;
        private final ByteBuf args;
```

```
        public Cmd(ByteBuf name, ByteBuf args) {
            this.name = name;
            this.args = args;
        }

        public ByteBuf name() {
            return name;
        }

        public ByteBuf args() {
            return args;
        }
    }

    public static final class CmdDecoder extends LineBasedFrameDecoder {
        public CmdDecoder(int maxLength) {
            super(maxLength);
        }

        @Override
        protected Object decode(ChannelHandlerContext ctx, ByteBuf buffer)
                throws Exception {
            ByteBuf frame =  (ByteBuf) super.decode(ctx, buffer);       #4
            if (frame == null) {
                return null;                                            #5
            }
            int index = frame.indexOf(frame.readerIndex(),
                    frame.writerIndex(), (byte) ' ');                   #6
            return new Cmd(frame.slice(frame.readerIndex(), index),
                    frame.slice(index +1, frame.writerIndex()));        #7
        }
    }

    public static final class CmdHandler
            extends SimpleChannelInboundHandler<Cmd> {
        @Override
        public void channelRead0(ChannelHandlerContext ctx, Cmd msg)
                throws Exception {
            // Do something with the command                            #8
        }
    }
}
```

#1 Add the CmdDecoder that will extract the command and forward to the next handler in the pipeline
#2 Add CmdHandler that will receive the commands
#3 The Pojo that represents the commands
#4 Extract the ByteBuf frame which was delimited by the \r\n
#5 There was no full frame ready so just return null
#6 Fix the first whitespace as it's the separator of the name and arguments
#7 Construct the command by passing in the name and the arguments using the index
#8 Handle the command

8.4.2 Length-based protocols

Often you find length-based protocols out in the wild. For this case, Netty ships with two different decoders that will help you extract frames, as shown in table 8.6.

Table 8.6 Decoders that extract frames based on the length

Name	Description
`FixedLengthFrameDecoder`	`Decoder` that extracts extra frames of the same fixed size.
`LengthFieldBasedFrameDecoder`	`Decoder` that extracts frames based on the size that's encoded in the header of the frame.

To make it clearer, let's illustrate both of them with a figure. Figure 8.6 shows how the `FixedLengthFrameDecoder` works.

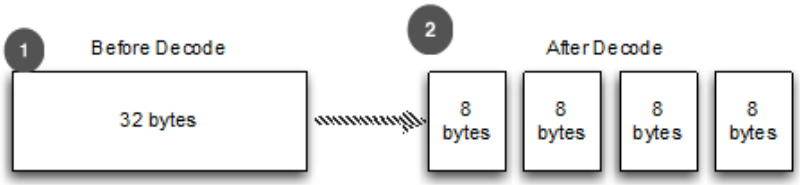


Figure 8.6 Message of fixed size of 8 bytes

#1 Stream of bytes
#2 Extracted frames of 8 byte size

As shown, the `FixedLengthFrameDecoder` extracts frames of a fixed length, which is 8 bytes in this case.

More often you find the case where the size of the frame is encoded in the header. For this purpose you can use the `LengthFieldBasedFrameDecoder`, which will read the length out of the header and extract the frame for the length.

Figure 8.7 shows how it works.

Figure 8.7 Message that has fixed size encoded in the header

#1 Length encoded in the header of the frame
#2 Content of the frame with the encoded length
#3 Extracted frame that has the length header stripped

If the length field is part of the header frame extracted, this can be configured using the constructor of the `LengthFieldBasedFrameDecoder`. Also, it lets you specify where exactly in the frame the length field is and how long it is. It's very flexible, so for more information, please refer to the API docs.

Because it's easy to use the `FixedLengthFrameDecoder`, we'll focus on the `LengthFieldBasedFrameDecoder`.

Listing 8.12 shows how you can use the `LengthFieldBasedFrameDecoder` to extract frames whose length is encoded in the first 8 bytes.

Listing 8.12 Decoder for the command and the handler

```
public class LengthBasedInitializer extends ChannelInitializer<Channel> {

    @Override
    protected void initChannel(Channel ch) throws Exception {
        ChannelPipeline pipeline = ch.pipeline();
        pipeline.addLast(
                new LengthFieldBasedFrameDecoder(65 * 1024, 0, 8));      #1
        pipeline.addLast(new FrameHandler());                            #2
    }

    public static final class FrameHandler
            extends SimpleChannelInboundHandler<ByteBuf> {
        @Override
        public void channelRead0(ChannelHandlerContext ctx,
                ByteBuf msg) throws Exception {
            // Do something with the frame                               #3
        }
    }
}
```

#1 Add LengthFieldBasedFrameDecoder to extract the frames based on the encoded length in the header
#2 Add FrameHandler to handle the frames
#3 Do something with the frames

8.5 Writing big data

Writing big chunks of data in an efficient way is often a problem when it comes to asynchronous frameworks, because you need to stop writing if the network is saturated or you'd otherwise fill up memory and have a good chance of getting an `OutOfMemoryError`.

Netty allows you to write the contents of a file using a zero-memory-copy approach, which means that it handles all the shuffling from the filesystem to the network stack in the kernel space and allows the maximum performance. This only works if the content of the Fille needs to be transfered as is (no in program modification); if not, Netty needs to copy the data into the user space to perform the operations on it. All of this happens in the core of Netty, so you don't need to worry about it.

Writing a file's contents via zero-memory-copy works by writing a `DefaultFileRegion` to the `Channel`, `ChannelHandlerContext`, or `ChannelPipeline`, as shown in listing 8.13.

Listing 8.13 Transfer file content with `FileRegion`

```
FileInputStream in = new FileInputStream(file);                          #1
FileRegion region = new DefaultFileRegion(
        in.getChannel(), 0, file.length());                              #2

channel.writeAndFlush(region)
        .addListener(new ChannelFutureListener() {                       #3
    @Override
    public void operationComplete(ChannelFuture future)
            throws Exception {
        if (!future.isSuccess()) {
            Throwable cause = future.cause();                            #4
            // Do something
        }
    }
});
```

#1 Get FileInputStream on file
#2 Create a new DefaultFileRegion for the file starting at offset 0 and ending at the end of the file
#3 Send the DefaultFileRegion and register a ChannelFutureListener
#4 Handle failure during send

But what do you do if you don't want to send a file but some other big chunk of data?

Netty ships with a special handler, called `ChunkedWriteHandler`, that allows you to write big chunks of data by handling `ChunkedInput` implementations. Table 8.7 shows these implementations.

Table 8.7 Provided ChunkedInput implementations

Name	Description
`ChunkedFile`	`ChunkedInput` implementation which allows you to write a file (use only if your platform doesn't support zero-memory-copy).
`ChunkedNioFile`	`ChunkedInput` implementation that allows you to write a file (use only if your platform doesn't support zero-memory-copy).
`ChunkedNioStream`	`ChunkedInput` implementation that allows you to transfer content from a `ReadableByteChannel`.
`ChunkedStream`	`ChunkedInput` implementation that allows you to transfer content from an `InputStream`.

The `ChunkedStream` implementation is the most used out there, so I'll show its use in listing 8.14.

Listing 8.14 Transfer file content with `FileRegion`

```
public class ChunkedWriteHandlerInitializer
        extends ChannelInitializer<Channel> {
    private final File file;

    public ChunkedWriteHandlerInitializer(File file) {
        this.file = file;
    }

    @Override
    protected void initChannel(Channel ch) throws Exception {
        ChannelPipeline pipeline = ch.pipeline();
        pipeline.addLast(new ChunkedWriteHandler());                    #1
        pipeline.addLast(new WriteStreamHandler());                     #2
    }

    public final class WriteStreamHandler
            extends ChannelInboundHandlerAdapter {

        @Override
        public void channelActive(ChannelHandlerContext ctx)
                throws Exception {
            super.channelActive(ctx);
            ctx.writeAndFlush(
                    new ChunkedStream(new FileInputStream(file)));    #3
        }}
}
```

#1 Add ChunkedWriteHandler to handle ChunkedInput implementations
#2 Add WriteStreamHandler to write a ChunkedInput
#3 Write the content of the file via a ChunkedStream once the connection is established (we use a FileInputStream only for demo purposes, any InputStream works)

8.6 Serializing data

When you want to transfer your POJOs over the network and act on them on the remote peer, Java offers the `ObjectOutputStream` and `ObjectInputStream` and its serialization marker interface. But there are also other options for doing this.

This section shows what Netty offers out of the box.

8.6.1 Serialization via plain JDK

If you need to talk with other peers that use `ObjectOutputStream` and `ObjectInputStream`, and you need to keep compatibility or you don't want to have an external dependency, JDK Serialization is the choice. Table 8.8 explains your options.

Table 8.8 Provided JDK Serialization codec

Name	Description
`CompatibleObjectDecoder`	Uses plain JDK Serialization for decoding and can be used with other peers that don't use Netty.
`CompatibleObjectEncoder`	Uses plain JDK Serialization for encoding and can be used

	with other peers that don't use Netty.
`CompactObjectDecoder`	Uses custom serialization for decoding on top of JDK Serialization. Only use this if you want to gain speed but not be able to have a dependency, in which case the other serialization implementations are preferable.
`CompactObjectEncoder`	Uses custom serialization for encoding on top of JDK Serialization. Only use this if you want to gain speed but not be able to have a dependency, in which case the other serialization implementations are preferable.

8.6.2 Serialization via JBoss Marshalling

If you can take an extra dependency, JBoss Marshalling is the way to go. It's up to three times faster than JDK Serialization and more compact.

Netty comes with either a compatible implementation that can be used with other peers using JDK Serialization or one that can be used for maximum speed if the other peers also use JBoss Marshalling. Table 8.9 explains the JBoss Marshalling codecs.

Table 8.9 JBoss Marshalling codec

Name	Description
`CompatibleMarshallingDecoder`	Uses JDK Serialization for decoding and so can be used with other peers that don't use Netty.
`CompatibleMarshallingEncoder`	Uses JDK Serialization for encoding and so can be used with other peers that don't use Netty.
`MarshallingDecoder`	Uses custom serialization for decoding.
`MarshallingEncoder`	Uses custom serialization for encoding.

Listing 8.15 Using JBoss Marshalling

```
public class MarshallingInitializer extends ChannelInitializer<Channel> {

    private final MarshallerProvider marshallerProvider;
    private final UnmarshallerProvider unmarshallerProvider;

    public MarshallingInitializer(UnmarshallerProvider unmarshallerProvider,
                                  MarshallerProvider marshallerProvider) {
        this.marshallerProvider = marshallerProvider;
        this.unmarshallerProvider = unmarshallerProvider;
    }
    @Override
    protected void initChannel(Channel channel) throws Exception {
```

```
        ChannelPipeline pipeline = channel.pipeline();
        pipeline.addLast(new MarshallingDecoder(unmarshallerProvider));
        pipeline.addLast(new MarshallingEncoder(marshallerProvider));
        pipeline.addLast(new ObjectHandler());
    }

    public static final class ObjectHandler
            extends SimpleChannelInboundHandler<Serializable> {
        @Override
        public void channelRead0(ChannelHandlerContext channelHandlerContext,
Serializable serializable) throws Exception {
            // Do something
        }
    }
}
```

#1 Add ChunkedWriteHandler to handle ChunkedInput implementations
#2 Add WriteStreamHandler to write a ChunkedInput
#3 Write the content of the file via a ChunkedStream once the connection is established (we use a FileInputStream only for demo purposes, any InputStream works)

8.6.3 Serialization via ProtoBuf

The last solution for serialization that ships with Netty is a codec that makes use of ProtoBuf.

ProtoBuf was open-sourced by Google in the past and is a way to encode and decode structured data in a compact and efficient way. It comes with different bindings for all sorts of programming languages, making it a good fit for cross-language projects. Table 8.10 shows the ProtoBuf implementations.

Table 8.10 Protobuf codec

Name	Description
`ProtobufDecoder`	Decode message via ProtoBufs.
`ProtobufEncoder`	Encode message via ProtoBufs.
`ProtobufVarint32FrameDecoder`	A decoder that splits the received `ByteBufs` dynamically by the value of the Google Protocol `BuffersBase` 128 Varints integer length field in the message.

Listing 8.16 Using Google Protobuf

```
public class ProtoBufInitializer extends ChannelInitializer<Channel> {

    private final MessageLite lite;

    public ProtoBufInitializer(MessageLite lite) {
        this.lite = lite;
    }

    @Override
    protected void initChannel(Channel ch) throws Exception {
        ChannelPipeline pipeline = ch.pipeline();
        pipeline.addLast(new ProtobufVarint32FrameDecoder());          #1
```

```
            pipeline.addLast(new ProtobufEncoder());                          #2
            pipeline.addLast(new ProtobufDecoder(lite));                      #3
            pipeline.addLast(new ObjectHandler());                            #4
        }

        public static final class ObjectHandler
                extends SimpleChannelInboundHandler<Object> {
            @Override
            public void channelRead0(ChannelHandlerContext ctx, Object msg)
                    throws Exception {
                // Do something with the object
            }
        }
    }
```

#1 Add ProtobufVarint32FrameDecoder to break down frames
#2 Add ProtobufEncoder to handle encoding of messages
#3 Add ProtobufDecoder that decodes to messages
#4 Add ObjectHandler to handle the decoded messages

As you see, using Protobuf is an easy task. Add the right handlers in the `ChannelPipeline` and you're ready to go.

8.7 Summary

This chapter gave you a brief overview of the included codecs and handlers that you can reuse in your application. This should help you to find your way through the provided pieces and prevent you from reinventing the wheel.

You also saw how you're able to combine different codecs/handlers to build up the logic that you need, and you learned how you can extend provided implementations to adjust them for the needed logic.

Another advantage to reusing what Netty provides is that those parts are well tested by many users and should be robust.

This chapter only covered the most-used codecs and handlers that are bundled with Netty. There are more; but to cover all of them would take a much bigger chapter. Please refer to Netty's API docs to get an overview of the rest of them.

In the next chapter we'll look at how you bootstrap your server and combine handlers to get things running.

9

Bootstrapping Netty applications

This chapter covers

- Bootstrapping clients and servers
- Bootstrapping clients from within a channel
- Adding `ChannelHandlers`
- Using `ChannelOptions` and attributes

In previous chapters you learned how to write your own `ChannelHandler` and codecs and add them to the `ChannelPipeline` of the `Channel`. Now the question is: How do you assemble all of this?

You can use bootstrapping. Netty provides you with an easy and unified way to bootstrap your servers and clients. What is bootstrapping, and how does it fit into Netty? Bootstrapping is the process by which you configure your Netty server and client applications. Bootstraps allow these applications to be easily reusable.

Bootstraps are available for both client and server Netty applications. The purpose of each is to simplify the process of combining all of the components we've discussed previously (channels, pipeline, handlers, and so on). Bootstraps also provide Netty with a mechanism which ties in these components and makes them all work in the background.

This chapter will look specifically at how the following pieces fit together in a Netty application:

- `EventLoopGroup` type
- `Channel` type
- Set `ChannelOptions`
- `ChannelHandler` that will be called once `Channel` is registered
- Specify special attributes that are added on the `Channel`

- Set local and remote address
- Bind and/or connect (depending on the type)

Once you know how to use the various bootstraps, you can use them to configure the server and client. You'll also learn when it makes sense to share a bootstrap instance and why, giving you the last missing piece to be able to assemble all the parts we learned about in previous chapters and allowing you to use Netty in your next application.

9.1 Different types of bootstrapping

Netty includes two different types of bootstraps. One is used for server-like channels that accept connections and create "child" channels for the accepted connections. The second is for "client-like channels" that don't accept new connections and process everything in the "parent" channel. Don't get confused by "client-like channels," as this is also true for connectionless transports, which don't have "child channels".

One of those situations is when handling `DatagramChannel` instances. Those are used for UDP, which is connectionless. In other words, due to the nature of UDP, when handling UDP data it's not necessary to have a channel per connection, as is the case with TCP connections. Since a channel isn't required per connection, protocol is said to be connectionless. This means that a single channel can process all the data and no parent-child channel relationship is required.

The two different bootstrap implementations extend from one super-class named `AbstractBootstrap`. Figure 9.1 shows the hierarchy.

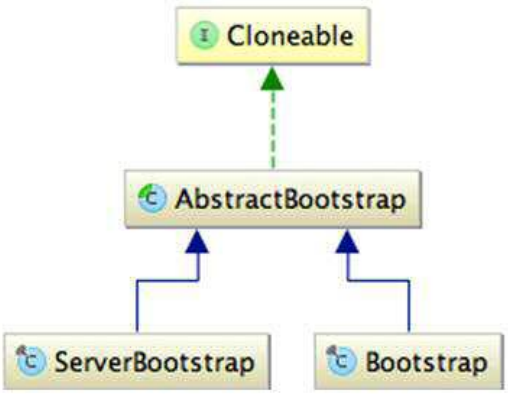


Figure 9.1 Bootstrap hierarchy

Many of the topics we've covered in previous chapters apply to both client and servers. In order to provide a common ground for the relationship between clients and servers, Netty uses the `AbstractBootstrap` class. By having a common ancestor, the client and server

bootstraps discussed in this chapter are able to reuse common functionality without duplicating code or logic.

It's often the case that multiple channels are needed with the same or very similar settings. Instead of creating a new bootstrap for each of these channels, Netty has made the `AbstractBootstrap` cloneable. This means that a deep clone of an already configured bootstrap will return another bootstrap which is reusable without having to be reconfigured. Netty's clone operation only makes a shallow copy of the bootstrap's `EventLoopGroup`, which means the `EventLoopGroup` is shared between all of the cloned channels. This is a good thing, as it's often the case that the cloned channels are short-lived, for example, a channel created to make an HTTP request.

The rest of the chapter will focus only on `Bootstrap` and `ServerBootstrap`. Let's have a stab at `Bootstrap` first, as it's not as complex as `ServerBootstrap`.

9.2 Bootstrapping clients and connectionless protocols

Whenever you need to bootstrap a client or some connectionless protocol you'll need to use the `Bootstrap` class. Working with it is easy, as we'll learn shortly. This section includes informations on the various methods for bootstrapping clients, explains how to bootstrap a client, and discusses how to choose a compatible channel implementation for the client.

9.2.1 Methods for bootstrapping clients

Before we go into much detail let's look at the various methods it provides. Table 9.1 gives an overview of them.

Table 9.1 Methods to bootstrap

Name	Description
`group(...)`	Set the `EventLoopGroup`, which should be used by the bootstrap. This `EventLoopGroup` is used to serve the I/O of the `Channel`.
`channel(...)` `channelFactory(...)`	The class of the `Channel` to instance. If the channel can't be created via a no-args constructor, you can pass in a `ChannelFactory` for this purpose.
`localAddress(...)`	The local address the `Channel` should be bound to. If not specified, a random one will be used by the operating system. Alternatively, you can specify the `localAddress` on `bind(...)` or `connect(...)`
`option(...)`	`ChannelOptions` to apply on the `Channel's ChannelConfig`. Those options will be set on the channel on the bind or connect method depending on what is called first. Changing them after calling those methods has no effect. Which `ChannelOptions` are supported

	depends on the actual channel you'll use. Please refer to the API docs of the `ChannelConfig` that's used by it.
`attr(...)`	Allow applying attributes on the channel. Those options will be set on the channel on the bind or connect method, depending on what is called first. Changing them after calling those methods has no effect.
`handler(...)`	Set the `ChannelHandler` that's added to the `ChannelPipeline` of the channel and so receive notification for events..
`clone()`	Clone the `Bootstrap` to allow it to connect to a different remote peer with the same settings as on the original `Bootstrap`.
`remoteAddress(...)`	Set the remote address to connect to. Alternatively, you can also specify it when calling `connect(...)`.
`connect(...)`	Connect to the remote peer and return a `ChannelFuture`, which is notified once the connection operation is complete. This can either be because it was successful or because of an error. Be aware that this method will also bind the Channel before.
`bind(...)`	Bind the channel and return a `ChannelFuture`, which is notified once the bind operation is complete. This can either be because it was successful or because of an error. Be aware that you will need to call `Channel.connect(…)` to finally connect to the remote peer after the bind operation success.

The next section explains how bootstrapping a client works and gives an example you can follow.

9.2.2 How to bootstrap a client

Now that you've been introduced to the different ways to bootstrap a client, I can show you how it works.

The bootstrap is responsible for client and/or connectionless-based channels, so it will create the channel after `bind(...)` or `connect(...)` is called.

Figure 9.2 shows how this works.

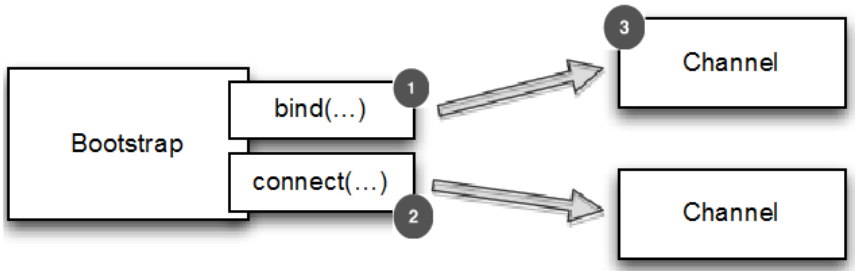


Figure 9.2 Bootstrap

#1 Bootstrap will create a new channel when calling bind(..) you will then later call connect(..) on the channel itself to establish the connection
#2 Bootstrap will create a new channel when calling connect(...)
#3 The newly created channel

Now that we know about all of the bootstrap methods, we'll look at how you use one in action. Listing 9.1 shows how you bootstrap a client that's using the NIO TCP transport.

Listing 9.1 Bootstrapping a client

```
Bootstrap bootstrap = new Bootstrap();                                  #1
bootstrap.group(new NioEventLoopGroup())                                #2
        .channel(NioSocketChannel.class)                                #3
        .handler(new SimpleChannelInboundHandler<ByteBuf>() {           #4
                 @Override
             protected void channeRead0(
                     ChannelHandlerContext channelHandlerContext,
                     ByteBuf byteBuf) throws Exception {
                        System.out.println("Reveived data");
                        byteBuf.clear();
                    }
                });
ChannelFuture future = bootstrap.connect(
        new InetSocketAddress("www.manning.com", 80));                  #5
    future.addListener(new ChannelFutureListener() {
        @Override
        public void operationComplete(ChannelFuture channelFuture)
                throws Exception {
            if (channelFuture.isSuccess()) {
                System.out.println("Connection established");
            } else {
                System.err.println("Connection attempt failed");
                channelFuture.cause().printStackTrace();
            }
        }
       });
```

#1 Create a new bootstrap to create new client channels and connect them
#2 Specify the EventLoopGroup to get EventLoops from and register with the channels
#3 Specify the channel class that will be used to instance
#4 Set a handler which will handle I/O and data for the channel
#5 Connect to the remote host with the configured bootstrap

As you may have noticed, all of the methods that don't finalize the bootstrap by either binding or connecting return a reference to the bootstrap itself. This allows for method chaining and gives you a DSL-like way to operate on it. If you remember, this is the same as what we did in the chapter on buffers. This is how Netty tries to keep things consistent in its API.

9.2.3 Choosing compatible channel implementations

The `Channel` implementation and the `EventLoop` that are processed by the `EventLoopGroup` must be compatible. Which channel is compatible to which `EventLoopGroup` can be found in the API docs. As a rule of thumb, you'll find the compatible pairs (`EventLoop` and `EventLoopGroup`) in the same package as the `Channel` implementation itself. For example, you'd use the `NioEventLoop`, `NioEventLoopGroup`, and `NioServerSocketChannel` together. Notice these are all prefixed with "Nio". You wouldn't, however, substitute any of these for another implementation with another prefix such as "Oio", that is, `OioEventLoopGroup` with `NioServerSocketChannel` would be incompatible, for example.

EventLoop and EventLoopGroup

Remember the EventLoop that is assigned to the Channel is responsible to handle all the operations for the Channel. Which means when-ever you execute a method that returns a ChannelFuture it will be executed in the EventLoop that is assigned to the Channel.

The EventLoop is executed by the Thread that is bound to it.

The EventLoopGroup contains a number of EventLoops and is responsible to assign an EventLoop to the Channel during it's registration.

If you try to use an incompatible `EventLoopGroup` it will fail, as shown in listing 9.2.

Listing 9.2 Bootstrap client with incompatible `EventLoopGroup`

```
Bootstrap bootstrap = new Bootstrap();                                  #1
bootstrap.group(new NioEventLoopGroup())                                #2
        .channel(OioSocketChannel.class)                                #3
        .handler(new SimpleChannelInboundHandler<ByteBuf>() {           #4
                @Override
                protected void channelRead0(
                    ChannelHandlerContext channelHandlerContext,
                            ByteBuf byteBuf) throws Exception {
                        System.out.println("Reveived data");
                        byteBuf.clear();
                    }
                });
ChannelFuture future = bootstrap.connect(
        new InetSocketAddress("www.manning.com", 80));                  #5
future.syncUninterruptibly();
```

#1 Create new bootstrap to create new client channels and connect them
#2 Specify the EventLoopGroup that will be used to get EventLoops from and register with the channels
#3 Specify the channel class that will be used to instance. You may notice that we use the Nio version for the EventLoopGroup and Oio for the channel
#4 Set a handler which will handle I/O and data for the channel
#5 Try to connect to the remote peer. This will throw an IllegalStateException as NioEventLoopGroup isn't compatible with OioSocketChannel

Setting up an implementation with an incompatible `EventLoopGroup` will ultimately fail with an `IllegalStateException`, as shown in listing 9.3.

Listing 9.3 `IllegalStateException` thrown because of invalid configuration

```
Exception in thread "main" java.lang.IllegalStateException: incompatible
event loop type: io.netty.channel.nio.NioEventLoop                    #1
     at
io.netty.channel.AbstractChannel$AbstractUnsafe.register(AbstractChannel.java
:571)
     at
io.netty.channel.SingleThreadEventLoop.register(SingleThreadEventLoop.java:57
)
     at
io.netty.channel.MultithreadEventLoopGroup.register(MultithreadEventLoopGroup
.java:48)
     at
io.netty.bootstrap.AbstractBootstrap.initAndRegister(AbstractBootstrap.java:2
98)
     at io.netty.bootstrap.Bootstrap.doConnect(Bootstrap.java:133)
     at io.netty.bootstrap.Bootstrap.connect(Bootstrap.java:115)
     at
com.manning.nettyinaction.chapter9.InvalidBootstrapClient.bootstrap(InvalidBo
otstrapClient.java:30)
     at
com.manning.nettyinaction.chapter9.InvalidBootstrapClient.main(InvalidBootstr
apClient.java:36)
     at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
     at
sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)
     at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.
java:43)
     at java.lang.reflect.Method.invoke(Method.java:601)
     at com.intellij.rt.execution.application.AppMain.main(AppMain.java:120)
```

#1 IllegalStateException tells you the configuration is not compatible

Use caution when constructing new bootstrap instances. Beside this there are other situations where an `IllegalArgumentException` will be thrown. This is when you haven't specified all the needed parameters before calling `bind(…)` or `connect(…)`.

The required parameters are:

- `group(...)`
- `channel(...)` or `channnelFactory(...)`

- `handler(...)`

Once these parameters are provided, your application can make full use of the Netty architecture as discussed so far. Pay special attention to the `handler(...)` method, as the channel pipeline needs to be configured appropriately as discussed in previous chapters.

9.3 Bootstrapping Netty servers with ServerBootstrap

After seeing how you can bootstrap clients and connectionless-based channels, it's time to see how to bootstrap a server. You'll see it's quite similar to and also shares some logic with bootstrapping clients. This section includes information on the various methods for bootstrapping servers and explains how to bootstrap a server.

9.3.1 Methods for bootstrapping servers

Again, let's first look over the provided methods of `ServerBootstrap`.

Table 9.2 Methods of ServerBootstrap

Name	Description
`group(...)`	Set the `EventLoopGroup`, which should be used by the `ServerBootstrap`. This `EventLoopGroup` is used to serve the I/O of the `ServerChannel` and accepted `Channels`.
`channel(...)` `channelFactory(...)`	The class of the `ServerChannel` to instance. If the channel can't be created via a no-args constructor, you can pass in a `ChannelFactory` for this purpose.
`localAddress(...)`	The local address the `ServerChannel` should be bound to. If not specified, a random one will be used by the operating system. Alternatively, you can specify the `localAddress` on `bind(...)` or `connect(...)`
`option(...)`	`ChannelOptions` to apply on the `ServerChannel` `ChannelConfig`. Those options will be set on the channel on the bind or connect method depending on what's called first. Changing them after calling those methods has no effects. Which `ChannelOptions` are supported depends on the actual channel you use. Please refer to the API docs of the `ChannelConfig` that is used.
`childOption(...)`	`ChannelOptions` to apply on the accepted `Channels` `ChannelConfig`. Those options will be set on the channels once accepted. Which `ChannelOptions` are supported depends on the actual channel you use. Please refer to the API docs of the `ChannelConfig` that is used.

`attr(...)`	Allow applying attributes on the `ServerChannel`. Those attributes will be set on the channel on the `bind(…)`. Changing them after calling `bind(…)` has no effects
`childAttr(...)`	Allow applying attributes on the accepted `Channels`. Those attributes will be set on the `Channel` once accepted. Changing them after calling those methods has no effects.
`handler(...)`	Set the `ChannelHandler` that is added to the `ChannelPipeline` of the `ServerChannel` and receive notification for events. You will not often specify one here. More important is the `childHandler(...)` when working with `ServerBootstrap`.
`childHandler(...)`	Set the `ChannelHandler` that's added to the `ChannelPipeline` of the accepted `Channels` and receive notification for events. The difference is between `handler(…)` and `childHandler(…)` is that `handler(…)` allows to add a handler which is processed by the "accepting" `ServerChannel`, while `childHandler(…)` allows to add a handler which processed by the "accepted" `Channel`. The accepted `Channel` represents here a bound Socket to a remote peer.
`clone()`	Clone the `ServerBootstrap` and allow using it to connect to a different remote peer with the same settings as on the original `ServerBoostrap`.
`bind(...)`	Bind the `ServerChannel` return to a `ChannelFuture`, which is notified once the connection operation is complete. This can either be because it was successful or because of an error.

The next section explains how bootstrapping a server works and gives an example you can follow.

9.3.2 How to bootstrap a server

As you may have noticed, the methods in the previous section are similar to what you saw in the bootstrap class.

There is only one difference, which makes a lot of sense once you think about it. While `ServerBootstrap` has `handler(...)`, `attr(...)`, and `option(...)` methods, it also offers those with the child prefix. This is done as the `ServerBootstrap` bootstraps

`ServerChannel` implementations, which are responsible for creating child channels. Those channels represent the accepted connections. `ServerBootstrap` offer the child* methods in order to make applying settings on accepted channels as easy as possible.

Figure 9.3 shows more detail about how the `ServerBootstrap` creates the `ServerChannel` on `bind(..)` and this `ServerChannel` manages the child `Channels`.

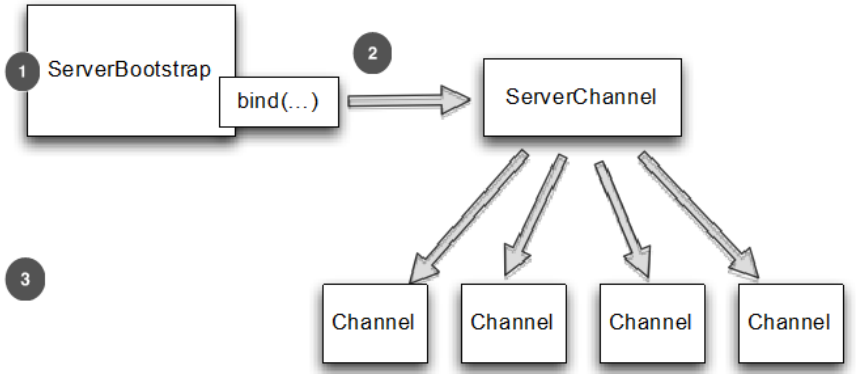


Figure 9.3 ServerBootstrap

#1 Bootstrap will create a new channel when calling bind(..) . This channel will then accept child channels once the bind is successful
#2 Accept new connections and create child channels that will serve an accepted connection
#3 Channel for an accepted connection

Remember the child* methods will operate on the child `Channels`, which are managed by the `ServerChannel`.

To make its use clearer, let's look at listing 9.4, which shows how to use `ServerBootstrap`, which will create a `NioServerSocketChannel` instance once `bind(...)` is called. This `NioServerChannel` is responsible for accepting new connections and creating `NioSocketChannel` instances for them.

Listing 9.4 Bootstrapping a server

```
ServerBootstrap bootstrap = new ServerBootstrap();                    #1
bootstrap.group(new NioEventLoopGroup(), new NioEventLoopGroup())     #2
        .channel(NioServerSocketChannel.class)                        #3
        .childHandler(new SimpleChannelInboundHandler<ByteBuf>() {    #4

           @Override
           protected void channelRead0(ChannelHandlerContext ctx,
                ByteBuf byteBuf) throws Exception {
                System.out.println("Reveived data");
                byteBuf.clear();
           }
    });
ChannelFuture future = bootstrap.bind(new InetSocketAddress(8080));   #5
```

```
future.addListener(new ChannelFutureListener() {

    @Override
    public void operationComplete(ChannelFuture channelFuture)
            throws Exception {
        if (channelFuture.isSuccess()) {
            System.out.println("Server bound");
        } else {
            System.err.println("Bound attempt failed");
            channelFuture.cause().printStackTrace();
        }
    }
    });
```

#1 Create a new ServerBootstrap to create new SocketChannel channels and bind them
#2 Specify the EventLoopGroups that will be used to get EventLoops from and register with the ServerChannel and the accepted channels
#3 Specify the channel class that will be used to instance
#4 Set a child handler which will handle I/O and data for the accepted channels
#5 Bind the channel as it's connectionless with the configured bootstrap

9.4 Bootstrapping clients from within a channel

Sometimes situations exist when you need to bootstrap a client `Channel` from within another `Channel`, for example, if you're writing a proxy or need to retrieve data from other systems. The case where you may need to fetch data from other systems is common since many Netty applications must integrate with an organization's existing systems. The integration could simply be to provide a point where the Netty application authenticates with an internal system and then queries a database.

Sure, you could create a new `Bootstrap` and use it as described in section 9.2.1. In fact, there's nothing wrong with that, it's just not as efficient as it could be, because you'll use another `EventLoop` for the newly created client channel, and if you need to exchange data between your accepted channel and the client channel you'll have to do context-switching between threads.

Fortunately Netty helps you optimize this by allowing you to pass in the `EventLoop` of the accepted channel to the `eventLoop(...)` method of the bootstrap, thereby allowing the client channel to operate on the same `EventLoop`. This eliminates all extra context switching and works because `EventLoop` extends `EventLoopGroup`.

Besides eliminating context-switching, you are able to use the bootstrap without needing to create more threads under the hood.

Why share the EventLoop ?

When you share the EventLoop you can be sure all Channels that are assigned to the EventLoop are using the same Thread. Remember an EventLoop is assigned to a Thread which execute the operations of it.

Because of using the same Thread there is no context-switching envolved and thus less overhead.

Figure 9.4 shows the relationship and how the same `EventLoop` will power both `Channels`.

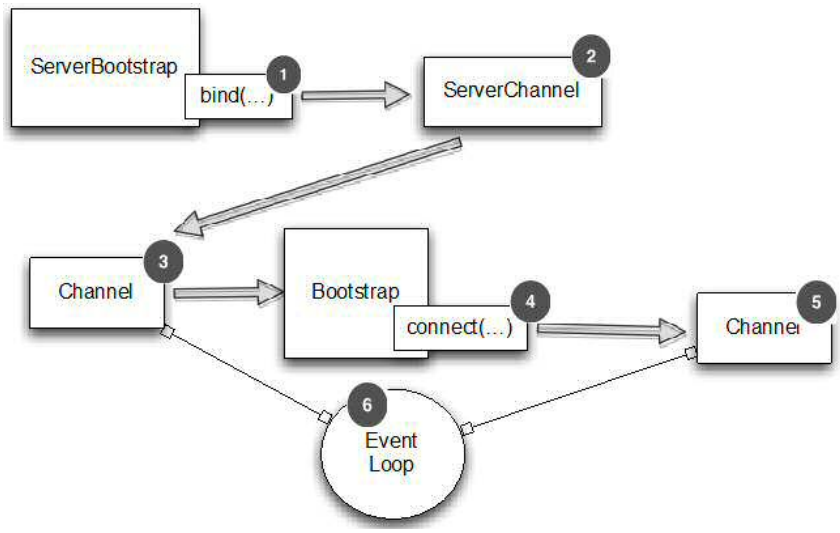


Figure 9.4 Share EventLoop between channels with ServerBootstrap and Bootstrap

#1 Bootstrap will create a new channel when calling bind(..) . This channel will then accept child channels once the bind is successful
#2 Accept new connections and create child channels that will serve a accepted connection
#3 Channel for an accepted connection
#4 Bootstrap which is created by the channel itself and create a new channel once the connect operation is called
#5 The newly created channel which is connected to the remote peer
#6 EventLoop which is shared between the channel which was created after accept and the new one that was created by connect

Sharing the EventLoop, as shown in figure 9.4, only needs some special attention while setting the EventLoop on the bootstrap via the Boostrap.eventLoop(. . .) method.

Listing 9.5 shows exactly what you need to do to handle this specific case.

Listing 9.5 Bootstrapping a server

```
ServerBootstrap bootstrap = new ServerBootstrap();                    #1
bootstrap.group(new NioEventLoopGroup(), new NioEventLoopGroup())     #2
        .channel(NioServerSocketChannel.class)                        #3
    .childHandler(new SimpleChannelInboundHandler<ByteBuf>() {        #4
        ChannelFuture connectFuture;

        @Override
```

```
        public void channelActive(ChannelHandlerContext ctx)
                throws Exception {
            Bootstrap bootstrap = new Bootstrap();                          #5
            bootstrap.channel(NioSocketChannel.class)                       #6
                     .handler(
                        new SimpleChannelInboundHandler<ByteBuf>() {        #7
                            @Override
                            protected void channelRead0(
                                    ChannelHandlerContext ctx,
                                    ByteBuf in) throws Exception {
                                System.out.println("Reveived data");
                                 in.clear();
                            }
                         });

            bootstrap.group(ctx.channel().eventLoop());                     #8
            connectFuture = bootstrap.connect(
                    new InetSocketAddress("www.manning.com", 80));          #9
            }

            @Override
            protected void channelRead0(ChannelHandlerContext
channelHandlerContext, ByteBuf byteBuf) throws Exception {
                if (connectFuture.isDone()) {
                    // do something with the data                          #10
                }
            }
        });

ChannelFuture future = bootstrap.bind(new InetSocketAddress(8080));         #11
future.addListener(new ChannelFutureListener() {
    @Override
    public void operationComplete(ChannelFuture channelFuture)
            throws Exception {
        if (channelFuture.isSuccess()) {
            System.out.println("Server bound");
        } else {
            System.err.println("Bound attempt failed");
            channelFuture.cause().printStackTrace();
        }
    }
});
```

#1 Create a new ServerBootstrap to create new SocketChannel channels and bind them
#2 Specify the EventLoopGroups to get EventLoops from and register with the ServerChannel and the accepted channels
#3 Specify the channel class which will be used to instance
#4 Set a handler which will handle I/O and data for the accepted channels
#5 Create a new bootstrap to connect to remote host
#6 Set the channel class
#7 Set a handler to handle I/O
#8 Use the same EventLoop as the one that's assigned to the accepted channel to minimize context-switching and so on
#9 Connect to remote peer
#10 Do something with the data if the connect is complete, for example, proxy it
#11 Bind the channel as it's connectionless with the configured bootstrap

This section showed how to reuse the `EventLoop`. Whenever possible, this resource should be reused in Netty applications. If `EventLoops` are not reused, take care to ensure that not too many instances are created, which could result in exhausting system resources.

9.5 Adding multiple ChannelHandlers during a bootstrap

In all of the code examples shown we only added one `ChannelHandler` during the bootstrap process by setting the instance via `handler(...)` or `childHandler(...)`. This may be good enough for simple applications but not for more complex ones. For example, in an application that must support multiple protocols, such as HTTP or WebSockets, a WebSocket fallback such as SockJS or Flash sockets would require a channel handler for each. Attempting to handle all these protocols in one channel handler would result in a large and complicated handler. Netty simplifies this by allowing you to provide as many handlers as are required.

One of Netty's strengths is its ability to "stack" many `ChannelHandlers` in the `ChannelPipeline` and write reusable code. But how do you do this if you can only set one `ChannelHandler` during the bootstrap process?

The answer is simple. Use only one, but use a special one . . .

For exactly this use case Netty provides a special abstract base class called `ChannelInitializer`, which you can extend to initialize a channel. This `ChannelHandler` will be called once the channel is registered on its `EventLoop` and allows you to add `ChannelHandlers` to the `ChannelPipeline` of the channel. This special initializer `ChannelHandler` will remove itself from the `ChannelPipeline` once it's done with initializing the channel.

Sounds complex? Not really. Once you've seen it in action in listing 9.6 it should seem as simple as it really is.

Listing 9.6 Bootstrap and using `ChannelInitializer`

```
ServerBootstrap bootstrap = new ServerBootstrap();                          #1
bootstrap.group(new NioEventLoopGroup(), new NioEventLoopGroup())           #2
    .channel(NioServerSocketChannel.class)                                  #3
    .childHandler(new ChannelInitializerImpl());                            #4

ChannelFuture future = bootstrap.bind(new InetSocketAddress(8080));         #5
future.sync();


final class ChannelInitializerImpl extends ChannelInitializer<Channel> {#6
    @Override
    protected void initChannel(Channel ch) throws Exception {
        ChannelPipeline pipeline = ch.pipeline();                           #7
        pipeline.addLast(new HttpClientCodec());
        pipeline.addLast(new HttpObjectAggregator(Integer.MAX_VALUE));

    }
}
```

#1 Create a new ServerBootstrap to create new Socketchannel channels and bind them
#2 Specify the EventLoopGroups which will be used to get EventLoops from and register with the ServerChannel and the accepted channels

#3 Specify the channel class which will be used to instance
#4 Set a handler for I/O and data for the accepted channels
#5 Bind the channel as it's connectionless with the configured bootstrap
#6 ChannelInitializer which is responsible to set up the ChannelPipeline
#7 Add needed handlers to ChannelPipeline. Once the initChannel(…) method completes the ChannelInitializer removes itself from the ChannelPipeline.

As mentioned before, more complex applications tend to require multiple channel handlers. By providing this special initializer as a handler, Netty allows you to insert as many channel handlers into the pipeline as your application requires.

9.6 Using Netty ChannelOptions and attributes

It would be annoying if you had to manually configure every channel when it's created. To avoid this, Netty allows you to apply what are called `ChannelOptions` to a bootstrap. These options are automatically applied to all channels created in the bootstrap. The various options available allow you to configure low-level details about connections such as the channel "keep-alive" or "timeout" properties.

Netty applications are often integrated with an organization's proprietary software. In some cases, Netty components, such as the channel, are passed around and used outside the normal Netty lifecycle. In cases like these, not all the usual properties and data are available. This is just one example, but for cases like these, Netty offers channel attributes.

Attributes allow you to associate data with channels in a safe way, and these attributes work just as well for both client and server channels.

For example, imagine a web server application where the client has made a request. In order to track which user a channel belongs to, the application can store the user's ID as an attribute of that channel. Similarly, any object or piece of data can be associated with a channel by using attributes.

Making use of the `ChannelOptions` and attributes is also simple and can make things a lot easier. For example, imagine a case where a Netty WebSocket server was automatically routing messages depending on the user. By using attributes, the application can store the user's ID with the channel to determine where a message should be routed. The application could be further automated by using a channel option which automatically terminates the connection if no messages have been received for routing within a given time. Listing 9.7 shows what happens when an invalid configuration is run.

Listing 9.7 `IllegalStateException` thrown due to invalid configuration

```
final AttributeKey<Integer> id = new AttributeKey<Integer>("ID");        #1

Bootstrap bootstrap = new Bootstrap();                                   #2
bootstrap.group(new NioEventLoopGroup())                                 #3
        .channel(NioSocketChannel.class)                                 #4
        .handler(new SimpleChannelInboundHandler<ByteBuf>()              #5
            @Override
            public void channelRegistered(ChannelHandlerContext ctx)
                    throws Exception {
                Integer idValue = ctx.channel().attr(id).get();          #6
```

```
                // do something  with the idValue
            }

            @Override
            protected void channelRead0(
                    ChannelHandlerContext channelHandlerContext,
                    ByteBuf byteBuf) throws Exception {
                System.out.println("Reveived data");
                byteBuf.clear();
            }
        });
bootstrap.option(ChannelOption.SO_KEEPALIVE,true)
        .option(ChannelOption.CONNECT_TIMEOUT_MILLIS, 5000);            #7
bootstrap.attr(id, 123456);                                             #8
ChannelFuture future = bootstrap.connect(
        new InetSocketAddress("www.manning.com", 80));                  #9
future.syncUninterruptibly();
```

#1 Create a new AttributeKey under which we'll store the attribute value
#2 Create a new bootstrap to create new client channels and connect them
#3 Specify the EventLoopGroup to get EventLoops from and register with the channels
#4 Specify the channel class that will be used to instance
#5 Set a handler which will handle I/O and data for the channel
#6 Retrieve the attribute with the AttributeKey and its value
#7 Set the ChannelOptions that will be set on the created channels on connect or bind
#8 Assign the attribute
#8 Connect to the remote host with the configured bootstrap

In the previous listings we used a `SocketChannel` in the bootstrap, which is TCP-based. As stated before, a bootstrap is also used for connectionless protocols such as UDP. For this Netty provides various `DatagramChannel` implementations. The only difference here is that you won't call `connect(...)` but only `bind(...)`, as shown in listing 9.8.

Listing 9.8 Using Bootstrap with `DatagramChannel`

```
Bootstrap bootstrap = new Bootstrap();
bootstrap.group(new OioEventLoopGroup()).channel(OioDatagramChannel.class)
       .handler(new SimpleChannelInboundHandler<DatagramPacket>(){

                @Override
            public void channelRead0(ChannelHandlerContext ctx,
                    DatagramPacket msg) throws Exception {
                // Do something with the packet
            }
        });
ChannelFuture future = bootstrap.bind(new InetSocketAddress(0));
future.addListener(new ChannelFutureListener() {
        @Override
        public void operationComplete(ChannelFuture channelFuture)
                throws Exception {
            if (channelFuture.isSuccess()) {
                System.out.println("Channel bound");
            } else {
                System.err.println("Bound attempt failed");
                channelFuture.cause().printStackTrace();
```

```
                }
            }
        });
```

#1 Create a new bootstrap to create new datagram channels and bind them
#2 Specify the EventLoopGroup to get EventLoops from and register with the channels
#3 Specify the channel class that will be used to instance
#4 Set a handler that will handle I/O and data for the channel
#5 Bind the channel as it's connectionless with the configured bootstrap

Netty comes with sane default configuration settings. In many cases you won't need to change these, but there are situations that demand absolute fine-grained control over how your application works and handles data. In these cases, Netty gives you the ability to provide these detailed configurations without too much effort.

9.7 Summary

In this chapter you learned how to bootstrap your Netty-based server and client implementation. You learned how you can specify configuration options that affect the and how you can use attributes to attach information to a channel and use it later.

You also learned how to bootstrap connectionless protocol-based applications and how they are different from connection-based ones.

The next chapters will focus on Netty in Action by using it to implement real-world applications. This will help you extract all interesting pieces for reuse in your next application.

At this point you should be able to start coding!

10 Unit-test your code

In this Chapter

- Unit testing
- EmbeddedChannel

As you learned in the previous chapters most of the time you implement one or more `ChannelHandler` for fulfill the various steps needed to actual handle received / send messages. But how to test that it works as expected and not break when refactor the code?

Fortunately testing your `ChannelHandler` implementations is quite easy as Netty offers you two extra classes, which can be used to do this.

After reading this Chapter you will be able to master the last missing piece of make your Netty application robust. Which is test your code…

As the shown tests use JUnit 4, a basic understanding of its usage is required. If you don't have it yet, take your time to read-up on using it. It's really simple but very powerful. You can find all needed information on the JUnit website. Beside this you also should have read the `ChannelHandler` and Codec chapters, as this Chapter will focus on testing your own implementation of a `ChannelHandler` or Codec.

10.1 General

As you learned before Netty offers an easy way to "stack" different `ChannelHandler` implementations together and so build up its `ChannelPipeline`. All of the `ChannelHandler` will then be evaluated while processing events.This design decision allows you to separate the logic into small reusable implementations, which only handle one task each. This not only makes the code cleaner but also allow to easier testing, which will be shown in this Chapter.

Testing of your `ChannelHandlers` is possible by using the "embedded" transport, which allows to easy pass events trough the pipeline and so test your implementations. For this the "embedded" transport offers a special`Channel` implementation, called `EmbeddedChannel`..

But how does it work? It's quite simple… The `EmbeddedChannel` allows you to write either inbound or outbound data into it and then check if something hit the end of the `ChannelPipeline`. This allows you to check if messages where encoded/decoded or triggered any action in your `ChannelHandler`.

Inbound vs Outbound

What is the difference between write inbound vs. outbound? Inbound data is processed by `ChannelInboundHandler` implementations and represent data that is read from the remote peer. Outbound data is processed by `ChannelOutboundHandler` implementations and represent data that will be written to the remote peer.

Thus depending on the `ChannelHandler` you want to test you would chose `writeInbound(...)` or `writeOutbound(…)` (or even both).

Those operations are provided by special methods and shown in Table 10.1.

Table 10.1 Special AbstractEmbeddedChannel methods

Name	Responsibility
`writeInbound(…)`	Write a message to the inbound of the `Channel`. This means it will get passed through the `ChannelPipeline` in the inbound direction. Returns `true` if something can now be read from the `EmbeddedChannel` via the `readInbound()` method.
`readInbound(…)`	Read a message from the `EmbeddedChannel`. Everything that will be returned here processed the whole `ChannelPipeline`. This method returns `null` if nothing is ready to read.
`writeOutbound(…)`	Write a message to the outbound of the `Channel`. This means it will get passed through the `ChannelPipeline` in the outbound direction. Returns `true` if something can now be read from the `EmbeddedChannel` via the `readOutbound()` method.
`readOutbound(…)`	Read a message from the `EmbeddedChannel`. Everything that will be

	returned here processed the whole `ChannelPipeline`. This method returns `null` if nothing is ready to read.
`finish()`	Mark the`EmbeddedChannel` as complete and return `true` if something can be read from either the inbound or outbound. This will also call close on the `Channel`.

To make it even more clear how the flow is let us have a look at Figure 10.1, which shows how messages / bytes will get passed through the `ChannelPipeline` by those explained methods.

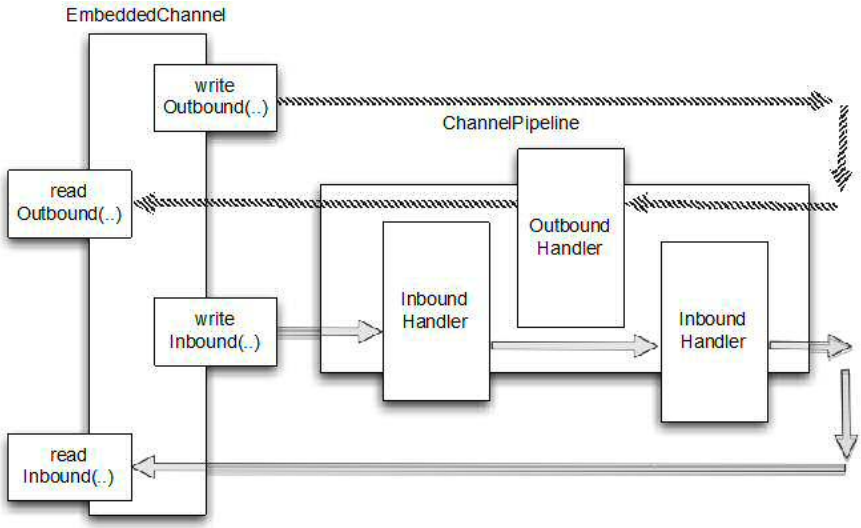


Figure 10.1 Message / byte flow

As shown in Figure 10.1 you can use the `writeOutbound(…)` method to write a message to the `Channel` and so let it pass trough the `ChannelPipeline` in the outbound direction. Later then you can read the processed message with `readOutbound(…)` and so see if the result is what you expect. The same is true for inbound, for which you can use `writeInbound(…)` and `readInbound(…)`. So the semantic is the same between handling outbound and inbound processing, it always traverses the whole `ChannelPipeline` and then is stored in the `EmbeddedChannel` if it hits the end.

With this general information it's time to have a detailed look at both of them and see how you can use each of them to test your logic.

10.2 Testing ChannelHandler

For testing `ChannelHandler` your best bet is to use `EmbeddedChannel`.

To illustrate how you can use `EmbeddedChannel` for testing a`ChannelHandler` let make use of it in this chapter and explain it with an example.

10.2.1 Testing inbound handling of messages

For this let us write a simple `ByteToMessageDecoder` implementation, which will produce frames of a fixed size once enough data can be read. If enough data isn't ready to be read it will wait for the next chunk of data and check again if a frame can be produced. And so on….

Figure 10.2 shows how received bytes will be re-assembled if it operates on frames with a fixed size of 3.

Figure 10.2Decoding via FixedLengthFrameDecoder

As you can see in Figure 10.3 it may consume more than one "event" to get enough bytes to produce a frame and pass it to the next `ChannelHandler` in the `ChannelPipeline`.

After the logic should be clear it's time for the implementation. It's shown in in Listing 10.1.

Listing 10.1 FixedLengthFrameDecoder implementation

```
public class FixedLengthFrameDecoder extends ByteToMessageDecoder {      #1

    private final int frameLength;

    public FixedLengthFrameDecoder(int frameLength) {                     #2
        if (frameLength <= 0) {
            throw new IllegalArgumentException(
                    "frameLength must be a positive integer: " +
frameLength);
        }
        this.frameLength = frameLength;
    }

    @Override
    protected void decode(ChannelHandlerContext ctx, ByteBuf in, List<Object>
out) throws Exception {
while(in.readableBytes() >= frameLength) {  #3
            ByteBuf buf = in.readBytes(frameLength);                      #4
            out.add(buf);                                                 #5
        }
    }
}
```

#1 Extend ByteToMessageDecoder and so handle inbound bytes and decode them to messages
#2 Specify the length of frames that should be produced
#3 Check if enough bytes are ready to read for process the next frame
#4 Read a new frame out of the ByteBuf
#5 Add the frame to the List of decoded messages.

Once the implementation is done it's always a good idea to write a unit test to make sure it works as expected. Even if you are sure you should write one just to guard yourself from problems later as you may refactor your code at some point. This way you will be able to spot problems before it's deployed in production.

Now let us have a look at how this is accomplished by using the `EmbeddedChannel`.

Listing 10.2 shows the test case.

Listing 10.2 Test the FixedLengthFrameDecoder

```
public class FixedLengthFrameDecoderTest {

    @Test                                                                #1
    public void testFramesDecoded() {
        ByteBuf buf = Unpooled.buffer();                                 #2
        for (int i = 0; i < 9; i++) {
            buf.writeByte(i);
        }
        ByteBuf input = buf.duplicate();

        EmbeddedChannel channel = new EmbeddedChannel(
new FixedLengthFrameDecoder(3));                                  #3
        // write bytes                                                   #4
        Assert.assertTrue(channel.writeInbound(input));

        Assert.assertTrue(channel.finish());                             #5

        // read messages                                                 #6
        Assert.assertEquals(buf.readBytes(3), channel.readInbound());
        Assert.assertEquals(buf.readBytes(3), channel.readInbound());
        Assert.assertEquals(buf.readBytes(3), channel.readInbound());
        Assert.assertNull(channel.readInbound());
    }


    @Test
    public void testFramesDecoded2() {
        ByteBuf buf = Unpooled.buffer();
        for (int i = 0; i < 9; i++) {
            buf.writeByte(i);
        }
        ByteBuf input = buf.duplicate();

        EmbeddedChannel channel = new EmbeddedChannel(new
FixedLengthFrameDecoder(3));
        Assert.assertFalse(channel.writeInbound(input.readBytes(2)));
        Assert.assertTrue(channel.writeInbound(input.readBytes(7)));

        Assert.assertTrue(channel.finish());
        Assert.assertEquals(buf.readBytes(3), channel.readInbound());
        Assert.assertEquals(buf.readBytes(3), channel.readInbound());
        Assert.assertEquals(buf.readBytes(3), channel.readInbound());
        Assert.assertNull(channel.readInbound());
    }
```

```
}
```

#1 Annotate with @Test to mark it as a test method
#2 Create a new ByteBuf and fill it with bytes
#3 Create a new EmbeddedByteChannel and feed in the FixedLengthFrameDecoder to test it
#4 Write bytes to it and check if they produced a new frame (message)
#5 Mark the channel finished
#6 Read the produced messages and test if its what was expected

To better understand what is done here let us have a deeper look at the concept and what is done. The `testFramesDecoded()` method wants to test a `ByteBuf`, which contains 9 readable bytes is decoded in 3 `ByteBuf`, which contains 3 readable bytes each. As you may notice it writes the `ByteBuf` with 9 readable bytes in one call of `writeInbound(…)` here. After this the `finish()` method is executed to mark the `EmbeddedByteChannel` as complete. Finally it calls `readInbound()` to read the produced frames out of the `EmbeddedByteChannel` until nothing can be read anymore.

The `testFramesDecoded2()` act the same way but with one difference. Here the inbound `ByteBufs` are written in two steps. When `writeInbound(input.readBytes(2))` is called, false is returned as the `FixedLengthFrameDecoder` will only produce output when at least 3 bytes are readable (as you may remember). The rest of the test works quite the same as `testFramesDecoded()`.

10.2.2 Testing outbound handling of messages

After you know all you need to test inbound handling of messages it's time to review how you would handle messages for outbound processing and test them the right way. The concept is quite the same as when handling inbound data, but let us review this while working through an example again.

To show how to test outbound byte processing let us implement the `AbsIntegerEncoder`. What it does is the following:

- Once a `flush(…)` is received it will read all ints from the `ByteBuf` and call `Math.abs(..)` on them
- Once done it write the bytes to the next `ByteBuf` in the `ChannelHandlerPipeline`.

Figure 10.3shows the logic.

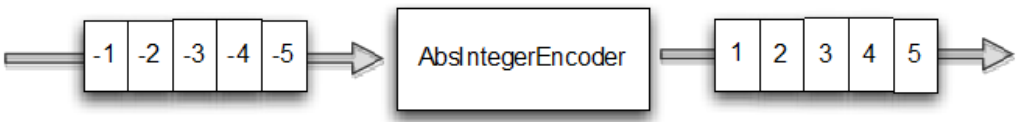


Figure 10.3 Encoding via AbsIntegerEncoder

The logic in Figure 10.4 also is the building ground for the actual implementation, which we will use to write our tests against.

Listing 10.3 shows it in detail.

Listing 10.3 AbsIntegerEncoder

```
public class AbsIntegerEncoder extends MessageToMessageEncode<ByteBuf>{
#1
    @Override
    protected void encode(ChannelHandlerContext channelHandlerContext,
ByteBuf in, List<Object>out) throws Exception {
        while (in.readableBytes() >= 4) {                                #2
            int value = Math.abs(in.readInt());                          #3
            out.add(value);      #4
        }
    }
}
```

#1 Extend MessageToMessageEncodeto encode a message to another message
#2 Check if there is enough bytes to encode
#3 Read the next int out of the input ByteBuf and calculate the absolute int out of it
#4 Write the intto the List of encoded messages

The implementation shown in Listing 10.3 is an exact implementation of the behavior described. So it works the same way as shown in Figure 10.4. Again we could just rely on the belief that it is working, but this is not what you want. A unit-test is the only way to know everything works.

As before we make use of `EmbeddedChannel` again to test the `AbsIntegerEncoder`.

Listing 10.4 Test the AbsIntegerEncoder

```
public class AbsIntegerEncoderTest {

    @Test                                                                #1
    public void testEncoded() {
        ByteBuf buf = Unpooled.buffer();                                 #2
        for (int i = 1; i < 10; i++) {
            buf.writeInt(i * -1);
        }

        EmbeddedChannel channel = new EmbeddedChannel(
new AbsIntegerEncoder());                                        #3
        Assert.assertTrue(channel.writeOutbound(buf));                   #4

        Assert.assertTrue(channel.finish());                             #5

        // read bytes                                                    #6
        ByteBuf output = (ByteBuf) channel.readOutbound();
        for (int i = 1; i < 10; i++) {
            Assert.assertEquals(i, output.readInt());
        }
        Assert.assertFalse(output.isReadable());
        Assert.assertNull(channel.readOutbound());
    }
}
```

#1 Annotate with @Test to mark it as a test method
#2 Create a new ByteBuf and fill it with bytes
#3 Create a new EmbeddedChannel and feed in the AbsIntegerEncoderto test it
#4 Write bytes to it and check if they produced bytes

#5 Mark the channel finished
#6 Read the produced messages and test if its what was expected

What the test code does is the following:

- Create a new `ByteBuf` which holds 10 ints
- Create a new `EmbeddedChannel`
- Write the `ByteBuf` to the outbound of the `EmbeddedChannel`. Remember `MessageToMessageEncoder` is a `ChannelOutboundHandler` which will manipulate data that should be written to the remote per, thus we need to use `writeOutbound(…)`.
- Mark the channel finish
- Read all ints out of the outbound output of the `EmbeddedChannel` and check if it only contain absolute ints

10.3 Testing exception handling

Sometimes transform inbound or outbound data is not enough; often you may need to also throw for example an Exception in some situations. This may be because you guard from malformed input or from handling to big resources or some other cause.

Again let us write an implementation, which throws a `TooLongFrameException` if the input bytes are more then a given limit. Such a feature is often used to guard against resource exhaustion.

Figure 10.4 shows how it will work if the frame size is limited to 3 bytes max.

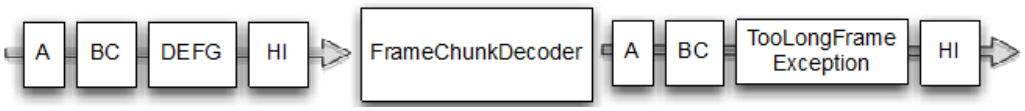


Figure 10.4 Decoding via FrameChunkDecoder

Figure 10.4 shows the logic. As soon as the input bytes exceed a limit, the bytes are discarded and a `TooLongFrameException` is thrown. The other `ChannelHandler` implementations in the `ChannelPipeline` can then handle the `TooLongFrameException` or just ignore it. The handling of the exception would be done in the `ChannelHandler.exceptionCaught(…)` method, for which the `ChannelHandler` implementation would provide some specific implementation.

The implementation is shown in Figure 10.5.

Listing 10.5 FrameChunkDecoder

```
public class FrameChunkDecoder extends ByteToMessageDecoder {              #1

    private final int maxFrameSize;

    public FrameChunkDecoder(int maxFrameSize) {
        this.maxFrameSize = maxFrameSize;
    }

    @Override
    protected void decode(ChannelHandlerContext ctx, ByteBuf in,
List<Object> out) throws Exception {
        int readableBytes = in.readableBytes();                             #2
        if (readableBytes > maxFrameSize)  {
            // discard the bytes                                            #3
            in.clear();
            throw new TooLongFrameException();
        }
        ByteBuf buf = in.readBytes(readableBytes);                          #4
        out.add(buf);                                                       #5
    }
}
```

#1 Extend ByteToMessageDecoder and so handle inbound bytes and decode them to messages
#2 Specify the max size of the frames that should be produced
#3 Discard the frame if its to big and throw a new TooLongFrameException which is an Exception provided by Netty and often used for this purpose.
#4 Read a new frame out of the ByteBuf
#5 Add the frame the List of decoded messages.

Again the best bet to test the code is to use `EmbeddedChannel`. The testing code is shown in Listing 10.6.

Listing 10.6 Testing FixedLengthFrameDecoder

```
public class FrameChunkDecoderTest {

    @Test                                                                   #1
    public void testFramesDecoded() {
        ByteBuf buf = Unpooled.buffer();                                    #2
        for (int i = 0; i < 9; i++) {
            buf.writeByte(i);
        }
        ByteBuf input = buf.duplicate();

        EmbeddedChannel channel = new EmbeddedChannel(
new FrameChunkDecoder(3));                                  #3

        Assert.assertTrue(channel.writeInbound(input.readBytes(2)));        #4
        try {
            channel.writeInbound(input.readBytes(4));                       #5
            Assert.fail();
        } catch (TooLongFrameException e) {
            // expected
```

```
        }
        Assert.assertTrue(channel.writeInbound(input.readBytes(3)));


        Assert.assertTrue(channel.finish());                                #6

        // Read frames                                                      #7
        Assert.assertEquals(buf.readBytes(2), channel.readInbound());
        Assert.assertEquals(buf.skipBytes(4).readBytes(3),
channel.readInbound());
    }
}
```

#1 Annotate with @Test to mark it as a test method
#2 Create a new ByteBuf and fill it with bytes
#3 Create a new EmbeddedByteChannel and feed in the FixedLengthFrameDecoder to test it
#4 Write bytes to it and check if they produced a new frame (message)
#5 Write a frame which is bigger then the max frame size and check if this cause an TooLongFrameException
#6 Mark the channel finished
#7 Read the produced messages and test if its what was expected

At first glance this looks quite similar to the test-code we had written in Listing 10.2. But one thing is "special" about it; the handling of the `TooLongFrameException`. After looking at Listing 10.6 you should notice the try / catch block which is used here. This is one of the "special" things of using the `EmbeddedChannel`. If one of the "write*" methods produce an Exception it will be thrown directly wrapped in a `RuntimeException`. This way it's easy to test if an Exception was thrown directly.

Even if we used the `EmbeddedChannel` with a `ByteToMessageDecoder`

It should be noted that the same can be done with any `ChannelHandler` implementation that throws an Exception.

10.4 Summary

In this chapter you learned how you are be able to test your custom `ChannelHandler` and so make sure it works like you expected. Using the shown techniques you are now be able to make use of JUnit and so ultimately test your code as your are used to.

Using the techniques shown in the chapter you will be able to guarantee a high quality of your code and also guard it from misbehavior..

In the next chapters we will focus on writing "real" applications on top of Netty and so show you how you can make real use of it. Even if the applications don't contain any test-code remember it is quite important to do so when you will write your next-gen application.

11 WebSockets

In this Chapter

- `WebSockets`
- `ChannelHandler, Decoder and Encoder`
- `Bootstrap an Netty based Application`
- `Test WebSockets`

The term "real-time-web" is everywhere these days. Most users expect to get information in real-time while visiting their website.

Netty comes with support for WebSockets bundled, which includes different versions. This makes it a no-brainer to use it in your next application. Because of this you will not need to worry about the protocol internals at all. Just use it in an easy way.

In this chapter you will develop an example application, which makes use of WebSockets and so help you to understand how you can make use of it. You will learn how you can integrate the shown parts in your application and so re-use some of them.

11.1 WebSockets some background

11.2 The Challenge

To show the "real-time" support of WebSockets, the application will show how you can use WebSockets in Netty to implement an IRC like chat application which can be used from within the browser. Kind of what you may know from Facebook where you can send text-messages to another person. But here we will go even further. In this application different users will be able to talk to each other at the same time. So really like IRC.

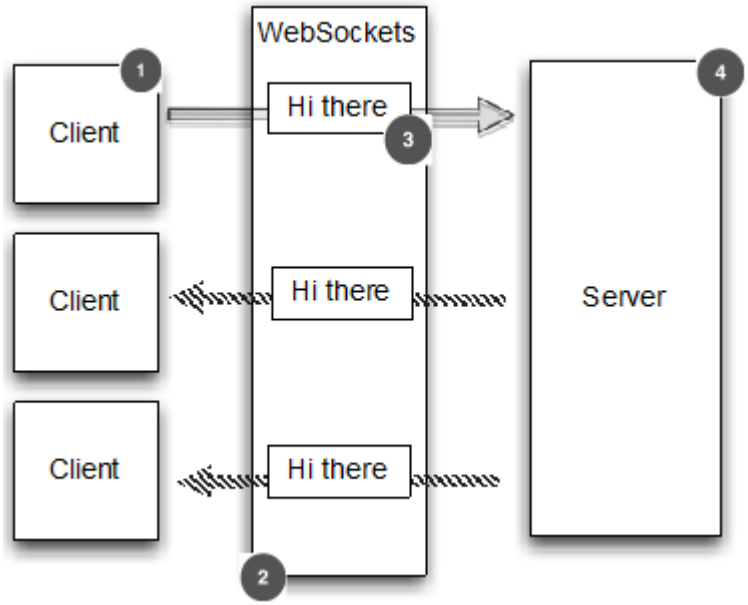


Figure 11.1 Application logic

#1 Client / User which connects to the Server and so is part of the chat
#2 Chat messages are exchanged of WebSockets
#3 Message send from Client to Server and then send from server to the clients
#4 The Server which handles all the Clients / Users

As shown in Figure 11.1 the logic is quite easy:

1 One client sends a message

2 The message is broadcasted to all other connected clients

It works just like how you would expect a chat-room to work. In this example we will only write the server part as the client will be a browser that accesses a web-page on which the chat-room will be displayed.

With this general idea in mind it's time to implement it now, which is quite straight forward, as you will see over the next pages.

11.3 Implementation

WebSockets uses the HTTP upgrade mechanism to switch from a "normal" HTTP(s) connection to WebSockets. Because of this an application, which uses WebSockets always starts with HTTP(s) and then perform the upgrade. When exactly the upgrade happens is specific to the application itself. Some directly perform the upgrade as first action some does it only after a specific url was requested.

In the case here we will only upgrade to WebSockets if the url ends with /ws otherwise the server will transmit a website to the client. Once upgraded the connection will transmit all data via WebSockets.

Figure 11.2 shows the logic.

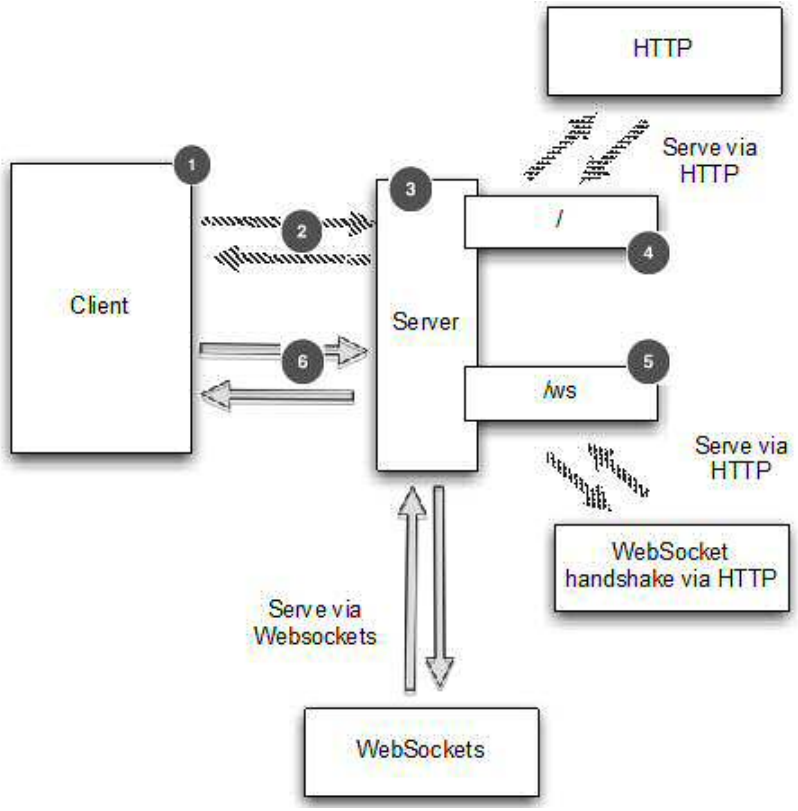


Figure 11.2 Server logic

#1 Client / User which connects to the Server and so is part of the chat
#2 Request Website or WebSocket handshake via HTTP
#3 The Server which handles all the Clients / Users
#4 Handles responses to the uri "/", which in this case is by send page the index.html page
#5 Handles the WebSockets upgrade if the uri "/ws" is accessed
#6 Send chat messages via WebSockets after the upgrade was complete

As always the logic will be implemented by different `ChannelHandlers` to make it reusable and easy to reuse later. The next section will give you more details about the `ChannelHandlers` that are needed and the various techniques they use for this purpose.

11.3.1 Handle http requests

As mentioned before the Server will act as a kind of hybrid to allow handling HTTP and WebSocket at the same time. This is needed as the application also needs to serve the html page, which is used to interact with the server and display messages that are sent via the chatroom.

Because of this we need write a `ChannelInboundHandler` that will handle `FullHttpRequests` if they are not sent to the /ws uri.

First have a look at Listing 11.1, which shows the implementation and then go in the details to explain all the used "features/techniques".

Listing 11.1 Handles HTTP requests

```
public class HttpRequestHandler
        extends SimpleChannelInboundHandler<FullHttpRequest> {          #1
    private final String wsUri;

    public HttpRequestHandler(String wsUri) {
        this.wsUri = wsUri;
    }

    @Override
    public void channelRead0(ChannelHandlerContext ctx, FullHttpRequest
request) throws Exception {
        if (wsUri.equalsIgnoreCase(request.getUri())) {
            ctx.fireChannelRead(request.retain());                      #2
        } else {
            if (HttpHeaders.is100ContinueExpected(request)) {
                send100Continue(ctx);                                   #3
            }

            RandomAccessFile file = new
RandomAccessFile("/path/to/index.html", "r");                           #4

            HttpResponse response = new DefaultHttpResponse(
                    request.getProtocolVersion(), HttpResponseStatus.OK);
            response.headers().set(
                    HttpHeaders.Names.CONTENT_TYPE,
                    "text/plain; charset=UTF-8");

            boolean keepAlive = HttpHeaders.isKeepAlive(request);

            if (keepAlive) {                                            #5
                response.headers().set(
                        HttpHeaders.Names.CONTENT_LENGTH, file.length());
                response.headers().set(
                        HttpHeaders.Names.CONNECTION,
                        HttpHeaders.Values.KEEP_ALIVE);
            }
            ctx.write(response);                                        #6

            if (ctx.pipeline().get(SslHandler.class) == null) {         #7
                ctx.write(new DefaultFileRegion(
```

```
                        file.getChannel(), 0, file.length()));
            } else {
                ctx.write(new ChunkedNioFile(file.getChannel());
            }
            ChannelFuture future = ctx.writeAndFlush(
                    LastHttpContent.EMPTY_LAST_CONTENT);                #8
            if (!keepAlive) {
                future.addListener(ChannelFutureListener.CLOSE);        #9
            }
        }
    }

    private static void send100Continue(ChannelHandlerContext ctx) {
        FullHttpResponse response = new DefaultFullHttpResponse(
                HttpVersion.HTTP_1_1, HttpResponseStatus.CONTINUE);
        ctx.writeAndFlush(response);
    }

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

#1 Extend SimpleChannelInboundHandler and handle FullHttpRequest messages
#2 Check if the request is an WebSocket Upgrade request and if so retain it and pass it to the next ChannelInboundHandler in the ChannelPipeline.
#3 Handle 100 Continue requests to conform HTTP 1.1
#4 Open the index.html file which should be written back to the client
#5 Add needed headers depending of if keepalive is used or not
#6 Write the HttpRequest to the client. Be aware that we use a HttpRequest and not a FullHttpRequest as it is only the first part of the request. Also we not use writeAndFlush(..) as this should be done later.
#7 Write the index.html to the client. Depending on if SslHandler is in the ChannelPipeline use DefaultFileRegion or ChunkedNioFile
#8 Write and flush the LastHttpContent to the client which marks the requests as complete.
#9 Depending on if keepalive is used close the Channel after the write completes.

The `HttpRequestHandler` shown in Figure 11.1 does the following:

- Check if the HTTP request was sent to the URI /ws and if so call `retain()` on the `FullHttpRequest` and forward it to the next `ChannelInboundHandler` in the ChannelPipeline by calling `ChannelHandlerContext.fireChannelRead(msg)`. The call of `retain()` is needed as after `channelRead0(…)` completes it will call `release()` on the `FullHttpRequest` and so release the resources of it. Remember this is how `SimpleChannelInboundHandler` works.
- Send a 100 Continue response if requested
- Write an `HttpResponse` back to the client after the headers are set.
- Create a new `DefaultFileRegion`, which holds the content of the index.html page and write it back to the client. By doing so zero-copy is archived which gives the best performance when transfer files over the network. Because this is only possible when

no encryption / compression etc is needed it checks if the SslHandler is in the ChannelPipeline and if so fallback to use ChunkedNioFile.

- Write a `LastHttpContent` to mark the end of the response and terminate it
- Add a `ChannelFutureListener` to the `ChannelFuture` of the last write if no keep-alive is used and so close the connection. The important thing to note here is that it called `writeAndFlush(…)` which means all previous written messages will also be flushed out to the remote peer.

But this is only one part of the application; we still need to handle the WebSocket frames, which will be used to transmit the chat messages. This will be covered in the next section.

11.3.2 Handle websocket frames

So we are already able to handle pure HTTP requests. Now it's time for some WebSockets action.

WebSockets support 6 different frames in the latest version. Table 11.1 shows all of them and also lists what they are used for

Table 11.1 WebSocketFrame types

Name	Description
`BinaryWebSocketFrame`	`WebSocketFrame` that contains binary data
`TextWebSocketFrame`	`WebSocketFrame` that contains text data
`ContinuationWebSocketFrame`	`WebSocketFrame` that contains text or binary data that belongs to a previous `BinaryWebSocketFrame` or `TextWebSocketFrame`
`CloseWebSocketFrame`	`WebSocketFrame` that represent a CLOSE request and contains close status code and a phrase
`PingWebSocketFrame`	`WebSocketFrame` which request the send of a `PongWebSocketFrame`
`PongWebSocketFrame`	`WebSocketFrame` which is send as response of a `PingWebSocketFrame`

For our application we are only interested in handling a few of them.

Those are:

- `CloseWebSocketFrame`
- `PingWebSocketFrame`
- `PongWebSocketFrame`
- `TextWebSocketFrame`

Fortunately we only need to explicit handle the `TextWebSocketFrame` as the others will automatically get handled by the `WebSocketServerProtocolHandler`. This makes things a lot easier for our implementation and us.

Listing 11.2 shows the needed handling of `TextWebSocketFrames`.

Listing 11.2 Handles Text frames

```
public class TextWebSocketFrameHandler extends
        ChannelInboundMessageHandlerAdapter<TextWebSocketFrame> {        #1
    private final ChannelGroup group;

    public TextWebSocketFrameHandler(ChannelGroup group) {
        this.group = group;
    }


    @Override
    public void userEventTriggered(ChannelHandlerContext ctx,
            Object evt) throws Exception {                               #2
        if (evt == WebSocketServerProtocolHandler
                .ServerHandshakeStateEvent.HANDSHAKE_COMPLETE) {
            ctx.pipeline().remove(HttpRequestHandler.class);             #3

            group.writeAndFlush(new TextWebSocketFrame("Client " +
                    ctx.channel() + " joined"));                         #4
          group.add(ctx.channel());                                      #5

        } else {
            super.userEventTriggered(ctx, evt);
        }
    }

    @Override
    public void channelRead0(ChannelHandlerContext ctx,
            TextWebSocketFrame msg) throws Exception {
        group.writeAndFlush(msg.retain());                               #6
    }
}
```

#1 Extend SimpleChannelInboundHandler and handle TextWebSocketFrame messages
#2 Override the userEventTriggered(...) method to handle custom events
#3 If the event is received that indicate that the handshake was successful remove the HttpRequestHandler from the ChannelPipeline as no further HTTP messages will be send.
#4 Write a message to all connected WebSocket clients about a new Channel that is now also connected
#5 Add the now connected WebSocket Channel to the ChannelGroup so it also receive all messages
#6 Retain the received message and write and flush it to all connected WebSocket clients.

The `TextWebSocketFrameHandler` shown in Figure 11.2 has again a very limited set of responsibilities, which are:

- Once the WebSocket handshake complete successfully write a message to the `ChannelGroup`, which holds all active WebSockets `Channels`. This means every

Channel in the `ChannelGroup` will received it After that add the Channel to the `ChannelGroup` as it is active now too

- If a `TextWebSocketFrame` is received, call `retain()` on it and write and flush it to the `ChannelGroup` so every connected WebSockets Channel gets it. Calling `retain()` is needed because other wise the `TextWebSocketFrame` would be released once the `channelRead0(…)` method returned. This is a problem as `writeAndFlush(…)` may complete later.

That's almost all you need as the rest is provided by Netty itself as stated before. So the only thing left is to provide a `ChannelInitializer` implementation, which will be used to initialize the `ChannelPipeline` for a new `Channel`.

11.3.3 Initialize the ChannelPipeline

Our last task is to initialize the `ChannelPipeline` with all the `ChannelHandlers` that are needed to get started for a new Channel / Connection. As in other chapters this is done by extending the `ChannelInitializer` class and implementing the `initChannel(…)` method.

Listing 11.3 shows it in detail

Listing 11.3 Init the ChannelPipeline

```
public class ChatServerInitializer extends ChannelInitializer<Channel> {#1
    private final ChannelGroup group;

    public ChatServerInitializer(ChannelGroup group) {
        this.group = group;
    }

    @Override
    protected void initChannel(Channel ch) throws Exception {           #2
    ChannelPipeline pipeline = ch.pipeline();
        pipeline.addLast(new HttpServerCodec());
        pipeline.addLast(new ChunkedWriteHandler());
        pipeline.addLast(new HttpObjectAggregator(64 * 1024));
        pipeline.addLast(new HttpRequestHandler("/ws"));
        pipeline.addLast(new WebSocketServerProtocolHandler("/ws"));
        pipeline.addLast(new TextWebSocketFrameHandler(group));
    }
}
```

#1 Extend ChannelInitializer as this should init the ChannelPipeline
#2 Add all needed ChannelHandler to the ChannelPipeline

As most of the times the **`initChannel(…)`** method setup the **`ChannelPipeline`** of the newly registered Channel. This is done by adding the different **`ChannelHandler`** to the **`ChannelPipeline`** that are needed to provide the logic needed.

Let us recap what handlers are added to the **`ChannelPipeline`** and what are their responsibilities in this case.

Table 12.1 Responsibilities of the ChannelHandlers

Name	Responsibility
`HttpServerCodec`	Decode bytes to HTTP requests / encode HTTP requests to bytes.
`ChunkedWriteHandler`	Allows to write a file content.
`HttpObjectAggregator`	Aggregate decoded `HttpRequest` / `HttpContent` / `LastHttpContent` to `FullHttpRequest`. This way you will always receive only "full" Http requests
`HttpRequestHandler`	Handle `FullHttpRequest` which are not send to /ws URI and so serve the index.html page
`WebSocketServerProtocolHandler`	Handle the WebSocket upgrade and Ping/Pong/Close WebSocket frames to be RFC compliant
`TextWebSocketFrameHandler`	Handles Text frames and handshake completion events

Something is special about the **`WebSocketServerProtcolHandler`** so it deserves a bit more of explanation. The **`WebSocketServerProtocol`** not only handles Ping/Pong/Close frames but also the handshake itself and so helps to "upgrade" to WebSockets.

This is done by execute the handshake and once successful modify the **`ChannelPipeline`** and so add all the needed encoders / decoders and also remove handlers that are not needed anymore.

To make it more clear have a look at Figure 11.3.

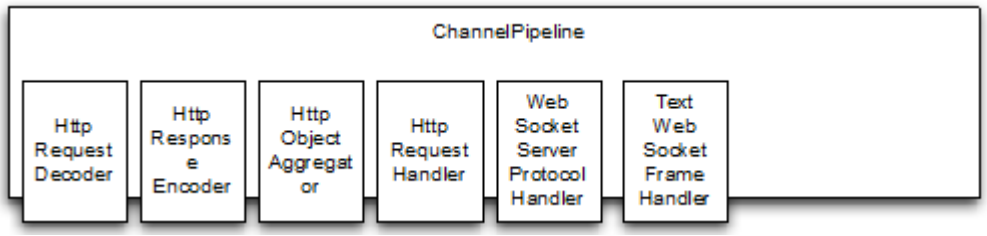


Figure 11.3 Server logic

Figure 11.3 shows the `ChannelPipeline` like it was initialized by the `initChannel(…)` method of `ChatServerInitializer`. But things change as soon as the handshake was completed. Once this is done the `WebSocketServerProtocolHandler` will take care of replace the `HttpRequestDecoder` with the WebSocketFrameDecoder13 and the `HttpResponseEncoder` with the WebSocketFrameEncoder13. Beside this it also remove all

"unnecessary" `ChannelHandlers` to gain optimal performance. Those are the `HttpObjectAggregator` and the `HttpRequestHandler`.

Figure 11.4 shows how the `ChannelPipeline` looks like the handshake completes.

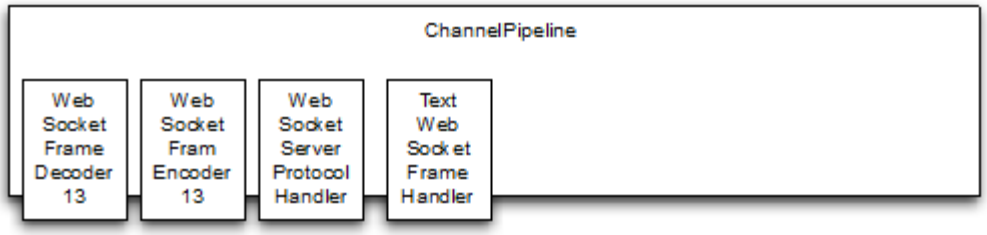


Figure 11.4 Server logic

This is done without you even noticing it as it is executed "under the hood". This is once again only possible because of the very flexible way of update the `ChannelPipeline` on the fly and separate tasks in different `ChannelHandler` implementations.

11.4 Wire things together

As always you need to wire things together to make use of them. As you learned before this is done via bootstraping the server and setting the correct **`ChannelInitializer`**.

Listing 11.4 shows the **`ChatServer`** class, which does all of this.

Listing 11.4 Bootstrap the server

```
public class ChatServer {

    private final ChannelGroup channelGroup =
        new DefaultChannelGroup(ImmediateEventExecutor.INSTANCE);       #1
    private final EventLoopGroup group = new NioEventLoopGroup();
    private Channel channel;

    public ChannelFuture start(InetSocketAddress address) {
        ServerBootstrap bootstrap  = new ServerBootstrap();             #2
        bootstrap.group(group)
                .channel(NioServerSocketChannel.class)
                .childHandler(createInitializer(channelGroup));
        ChannelFuture future = bootstrap.bind(address);
        future.syncUninterruptibly();
        channel = future.channel();
        return future;
    }

    protected ChannelInitializer<Channel> createInitializer(
            ChannelGroup group) {                                       #3
        return new ChatServerInitializer(group);
    }
```

```
    public void destroy() {                                                    #4
        if (channel != null) {
            channel.close();
        }
        channelGroup.close();
        group.shutdownGracefully();
    }

    public static void main(String[] args) {
        if (args.length != 1) {
            System.err.println("Please give port as argument");
            System.exit(1);
        }
        int port = Integer.parseInt(args[0]);

        final ChatServer endpoint = new ChatServer();
        ChannelFuture future = endpoint.start(new InetSocketAddress(port));

        Runtime.getRuntime().addShutdownHook(new Thread() {
            @Override
            public void run() {
                endpoint.destroy();
            }
        });
        future.channel().closeFuture().syncUninterruptibly();
    }
}
```

#1 Create DefaultChannelGroup which will hold all connected WebSocket clients
#2 Bootstrap the server
#3 Create the to be used ChannelInitializer
#4 Handle destroy of the Server and so release all resources

That's it. It's ready now for a test-run.

The easiest way to start up the application is to use the source-code provided for this book. It has everything ready to use and will use maven automatically to setup everything and download dependencies. Starting it up via mvn is as easy as shown in Listing 11.5

Listing 11.5 Compile and start ChatServer with maven

```
Normans-MacBook-Pro:netty-in-action-privatenorman$ mvn clean package
exec:exec -Pchapter11-ChatServer
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------
-----------
[INFO] Building netty-in-action 0.1-SNAPSHOT
[INFO] ------------------------------------------------------------------
-----------
...
...
...
[INFO]
```

```
    [INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ netty-in-action
    ---
    [INFO] Building jar: /Users/norman/Documents/workspace-
    intellij/netty-in-action-private/target/netty-in-action-0.1-
    SNAPSHOT.jar
    [INFO]
[INFO] --- exec-maven-plugin:1.2.1:exec (default-cli) @ netty-in-action
---
```

Now you can point your web-browser to **http://localhost:9999** and access the application. Figure 11.5 shows the interface included in the code for this chapter.

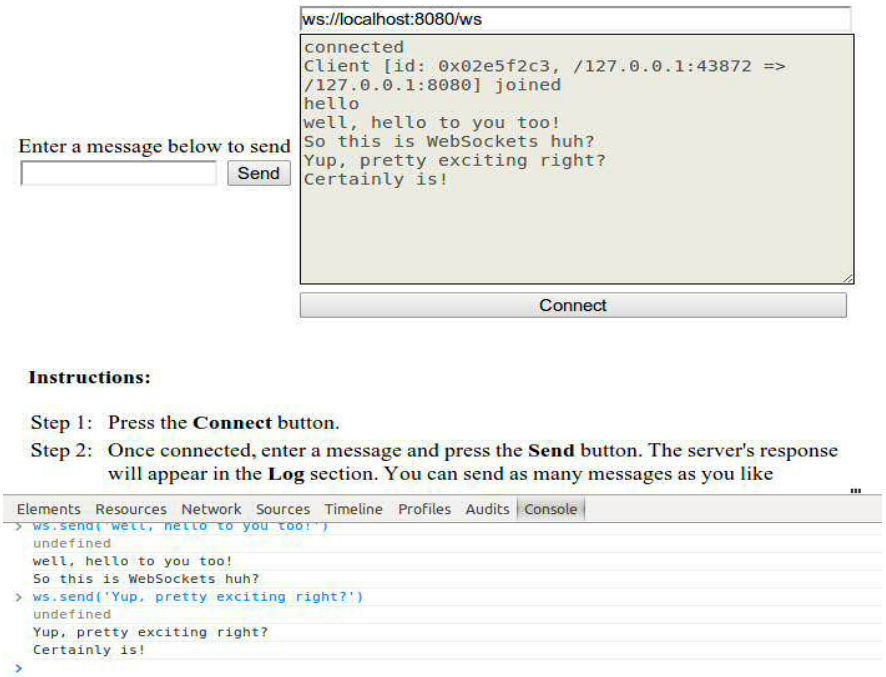


Figure 11.5 – WebSocket chat demonstration

In figure 11.5 above, we have two clients connected. The first client is connected using the interface at the top. The second client is connected via the browser's command line at the bottom. You'll notice that there are messages sent from both clients and each message is displayed to both. This is a very simple example of real-time communication that WebSocket enables in the browser.

11.5 We need encryption, now!

Even as the application works quite well you may notice at some point that requirements change. A typical change is the requirement of encryption, as you start to get concerned about transmitted data.

Often adding this kind of new feature is not an easy task, and need big changes to the project. But not when using Netty! It's as easy as add the **`SslHandler`** to the **`ChannelPipeline`** and that's it. So more or less a "one-line" change if you ignore the fact that you also need to configure the **`SslContext`** somehow. But this is not in scope of this example.

There is not much you need to adjust to make it use of encryption. Thanks again to the design of Netty's **`ChannelPipeline`**.

Let us have a look what changes are needed. First of it's needed to add a **`SslHandler`** to the **`ChannelPipeline`**. Listing 11.6 extends the previous created **`ChatServerInitializer`** to make this happen.

Listing 11.6 Init the ChannelPipeline with encryption

```
public class SecureChatServerIntializer extends ChatServerInitializer { #1
    private final SSLContext context;

    public SecureChatServerIntializer(ChannelGroup group, SSLContext
context) {
        super(group);
        this.context = context;
    }

    @Override
    protected void initChannel(Channel ch) throws Exception {
        super.initChannel(ch);
        SSLEngine engine = context.createSSLEngine();
        engine.setUseClientMode(false);
        ch.pipeline().addFirst(new SslHandler(engine));                #2
    }
}
```

#1 Extend the ChatServerInitializer as we want to enhance it's functionality
#2 Add the SslHandler to the ChannelPipeline to enable encryption

That's basically all what is needed. Now the only thing left is to adjust the `ChatServer` class to use the `SecureChatServerInitializer` and pass in the `SSLContext` to use. Listing 11.7 shows how this is done.

Listing 11.7 Init the ChannelPipeline with encryption

```
public class SecureChatServer extends ChatServer {                    #1

    private final SSLContext context;

    public SecureChatServer(SSLContext context) {
        this.context = context;
```

```
    }

    @Override
    protected ChannelInitializer<Channel> createInitializer(
            ChannelGroup group) {
        return new SecureChatServerInitializer(group, context);         #2
    }

    public static void main(String[] args) {
        if (args.length != 1) {
            System.err.println("Please give port as argument");
            System.exit(1);
        }
        int port = Integer.parseInt(args[0]);

        SSLContext context = BogusSslContextFactory.getServerContext();
        final SecureChatServer endpoint = new SecureChatServer(context);
        ChannelFuture future = endpoint.start(new InetSocketAddress(port));

        Runtime.getRuntime().addShutdownHook(new Thread() {
            @Override
            public void run() {
                endpoint.destroy();
            }
        });
        future.channel().closeFuture().syncUninterruptibly();
    }
}
```

#1 Extend ChatServer to enhance the functionality
#2 Return the previous created SecureChatServerInitializer to enable encryption

That's all the magic needed. The application is now using SSL/TLS to encrypt the network communication

As before you can use maven to startup the application and have it pull in all needed dependencies. Listing 11.8 shows how to start it.

Listing 11.8 Compile and start SecureChatServer with maven

```
Normans-MacBook-Pro:netty-in-action-private$ mvn clean package
exec:exec -Pchapter11-SecureChatServer
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------
-----------
[INFO] Building netty-in-action 0.1-SNAPSHOT
[INFO] ------------------------------------------------------------
-----------
...
...
...
[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ netty-in-action
---
```

```
    [INFO] Building jar: /Users/norman/Documents/workspace-
    intellij/netty-in-action-private/target/netty-in-action-0.1-
    SNAPSHOT.jar
    [INFO]
[INFO] --- exec-maven-plugin:1.2.1:exec (default-cli) @ netty-in-action
---
```

After it starts up you can access it via **https://localhost:9999**. Note that we use https now which means the network communication is encrypted and so safe to use.

11.6 Summary

In this chapter you learned how you could make use of WebSockets in a Netty based Application to use it for real-time data in the web.

You learned what types of data you can transmit and how you can make use of them. Beside this you should now have an idea about why WebSockets is one of the "new big things" out there.

Also you got a short overview about limitations which you may see while use WebSockets and why it may not be possible to use it in all the cases.

In the next chapter you will learn about another Web 2.0 thing, which may help you to make your service more appealing. The chapter will show you what all the hype of SPDY is about and why you may want to support it in your next application.

12 SPDY

In this Chapter we're going to look at

- An overview of SPDY in general
- ChannelHandler, Decoder and Encoder
- Bootstrap a Netty based Application
- Test SPDY / HTTP(s)

Netty comes bundled with support for SPDY and so makes it a piece of cake to have your application make use of it without worrying about all the protocol internals at all.

After this chapter you will know what you need to do to make your application "SPDY-ready" and also know how to support SPDY and HTTP at the same time.

This Chapter contains everything you need to know to get the example application working with SPDY and HTTP at the same time. While it can't harm to have read at least the `ChannelHandler` and Codec Chapter it is not needed as everything is provided here. However, this chapter will not go into all the details that are covered in the previous chapters.

12.1 SPDY some background

SPDY was developed by Google to solve their scaling issues. One of its main tasks is to make the "loading" of content as fast as possible.

For this SPDY does a few things:

- Every header that is transferred via SDPY gets compressed via GZIP. Compression the body is optional as it can be problematic for Proxy servers.
- Everything is encrypted via TLS
- Multiple transfer per connection possible
- Allows you to set priority for different data enabling critical files to be transferred first

Table 12.1 shows how this compares to HTTP.

Table 12.1 Compare HTTP and SPDY

Browser	HTTP 1.1	SPDY
Encrypted	Not by default	Yes
Header Compression	No	Yes
Full-Duplexing	No	Yes
Server-Push	No	Yes
Priorities	No	Yes

Some benchmarks have shown that a speed up of 50% percent is possible while using SPDY over HTTP.

While SPDY was only supported by Googe Chrome 2 years ago it is now supported in most of the Web-browsers at the time of writing.

SPDY is currently available in 3 different versions based on Drafts:

- Draft 1
- Draft 2
- Draft 3

Netty supports Draft 2 and 3 at the moment. Those are the drafts that are supported by the well-known Browsers.

Table 12.1 shows the browsers that support SPDY on the time of writing.

Table 12.1 Browsers

Browser	Version
Chrome	19+
Chromium	19+
Mozilla Firefox	11+ (enabled by default since 13)
Opera	12.10+

12.2 The Challenge

To keep it simple we will write a very simple application, which will show you, how you can integrate SPDY in your next application.

The application does only do one thing. It serves some static content. Based on whether its standard HTTPS or SPDY it will write different content to the client back. The switch to SPDY will get done automatically based on if the client (browser) supports the SPDY version, which is provided by our server.

Figure 12.1 shows the flow.

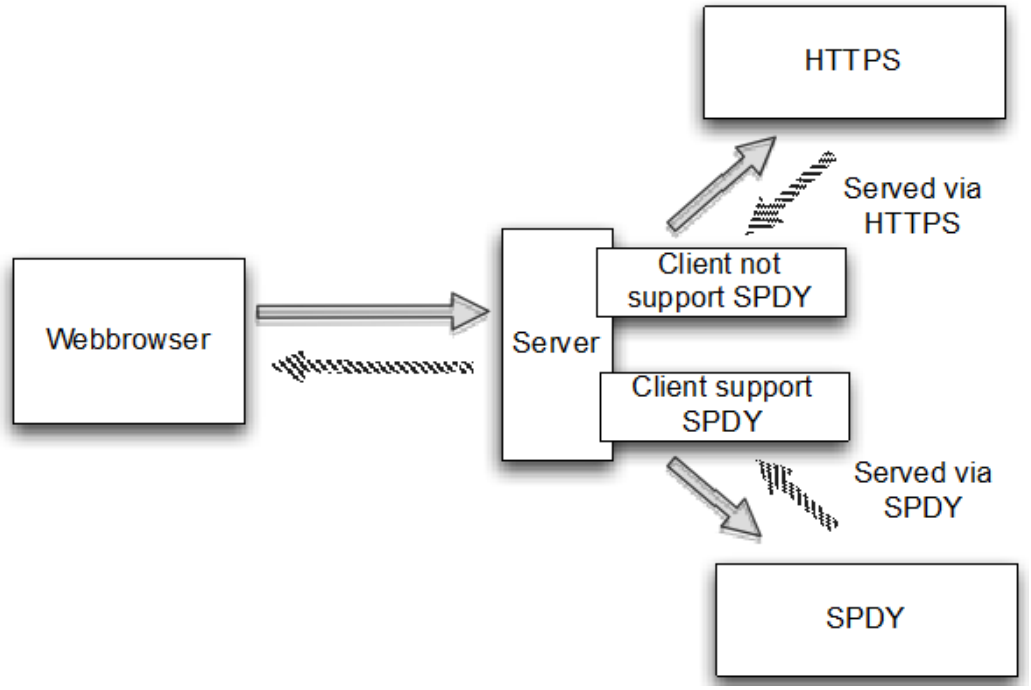


Figure 12.1 Application logic

For this application we will only write the server component, which handles HTTPS and SPDY. For the client side we will use a Web-browser, one that supports SPDY and one that does not, in order to show the difference.

12.3 Implementation

SPDY uses a TLS-Extension, which is called Next Protocol Negotiation (NPN) to choose the protocol. Using Java there are currently two different ways that allows hooking in here.

1. Use ssl_npn
2. Use NPN via the Jetty extension library

In this example we will use the jetty library to make use of SPDY. If you want to use ssl_npn please refer to the documentation of the project.

Jetty NPN Library

The Jetty NPN Library is an external library and so not part of Netty itself. It is responsible for the Next Protocol Negotiation, which is used to detect if the client supports SPDY or not.

12.3.1 Integration with Next Protocol Negotiation

The jetty library provides an interface called `ServerProvider`, which is used to determine the protocol to use. The `ServerProvider` implementation will then allow hooking in into the protocol selection. Depending on which HTTP versions and which SPDY versions should be supported it may differ in terms of implementation.

Listing 12.1 shows the implementation used by this example.

Listing 12.1 Implementation of ServerProvider

```
public class DefaultServerProvider implements NextProtoNego.ServerProvider {
    private static final List<String> PROTOCOLS =
            Collections.unmodifiableList(
Arrays.asList("spdy/2", "spdy/3", "http/1.1"));     #1

    private String protocol;

    @Override
    public void unsupported() {
        protocol = "http/1.1";                                          #2
    }

    @Override
    public List<String> protocols() {
        return PROTOCOLS;                                               #3
    }

    @Override
    public void protocolSelected(String protocol) {
        this.protocol = protocol;                                       #4
    }

    public String getSelectedProtocol() {
        return protocol;                                                #5
    }
}
```

#1 Define all supported protocols by this ServerProvider implementation
#2 Set the protocol to http/1.1 if SPDY detection failed
#3 Return all the supported protocols by this ServerProvider
#4 Set the selected protocol
#5Return the previous selected protocol

For this application we want to support 3 different protocols, and so have make them selectable in the implementation of our `ServerProvider`.

Those are:

- SPDY draft 2
- SPDY draft 3
- HTTP 1.1
-

If it fails to detect the protocol it will use HTTP 1.1 as default and so allow serving all clients even if SPDY isn't supported.

This is the only integration code that needs to get written for make use of NPN. Next on the list is to write the actual code to make use of it.

12.3.2 Implementation of the various ChannelHandlers

Now it's time to write the `ChannelInboundMessageHandlerAdapter`, which will handle the HTTP requests for the clients that do not support SPDY.

As mentioned before SPDY is still considered very new and so not all clients support it yet. For exactly this reason you should always provide the ability to fallback to HTTP as otherwise you will give many users a bad experience while using your service.

Listing 12.2 shows the handler, which handles HTTP traffic

Listing 12.2Implementation which handles HTTP

```
@ChannelHandler.Sharable
public class HttpRequestHandler
        extends SimpleChannelInboundHandler<FullHttpRequest> {
    @Override
    public void channelRead0(ChannelHandlerContext ctx,
FullHttpRequest request) throws Exception {                  #1
        if (HttpHeaders.is100ContinueExpected(request)) {
            send100Continue(ctx);                                          #2
        }

        FullHttpResponse response = new DefaultFullHttpResponse(
                request.getProtocolVersion(), HttpResponseStatus.OK);    #3

response.content().writeBytes(getContent()
.getBytes(CharsetUtil.UTF_8));                               #4
        response.headers().set(HttpHeaders.Names.CONTENT_TYPE,
 "text/plain; charset=UTF-8");                                #5

        boolean keepAlive = HttpHeaders.isKeepAlive(request);

        if (keepAlive) {                                                   #6
            response.headers().set(HttpHeaders.Names.CONTENT_LENGTH,
 response.content().readableBytes());
            response.headers().set(HttpHeaders.Names.CONNECTION,
 HttpHeaders.Values.KEEP_ALIVE);
        }
        ChannelFuture future = ctx.writeAndFlush(response);              #7

        if (!keepAlive) {
            future.addListener(ChannelFutureListener.CLOSE);             #8
        }
    }

    protected String getContent() {                                      #9
        return "This content is transmitted via HTTP\r\n";
    }

    private static void send100Continue(ChannelHandlerContext ctx) {    #10
        FullHttpResponse response = new DefaultFullHttpResponse(
```

```
HttpVersion.HTTP_1_1, HttpResponseStatus.CONTINUE);
        ctx.writeAndFlush(response);
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx,
Throwable cause)throws Exception {                          #11
        cause.printStackTrace();
        ctx.close();
    }
}
```

#1 Override the messageReceived(…) method which will be called for each FullHttpRequest that is received
#2 Check if a Continue response is expected and if so write one
#3 Create a new FullHttpResponse which ill be used as "answer" to the request itself
#4 Generate the content of the response and write it to the paylod of it
#5 Set the header so the client knows how to interpret the payload of the response
#6 Check if the request was set with the keepalive enabled, if so set the needed headers to fulfill the http rfc
#7 Write the response back to the client and get a reference to the future which will get notified once the write completes
#8 If the response is not using keepalive take care of close the connection after the write completes
#9 Return the content which will be used as payload of the response
#10 Helper method to generate the 100 continue response and write it back to the client
#11 Close the channel if an exception was thrown during processing

Let us recap what the implementation in Listing 12.2 actual does:

1. The `channelRead0(….)` method is called once a `FullHttpRequest` was received.
2. A `FullHttpResponse` with status code 100 (continue) is written back to the client if expected
3. A new `FullHttpResponse` is created with status code 200 (ok) and its payload is filled with some content
4. Depending on if keep-alive is set in the `FullHttpRequest`, headers are set to the `FullHttpResponse` to be RFC compliant.
5. The `FullHttpResponse` is written back to the client and the channel (connection) is closed if keep-alive is not set

This is how you typically would handle HTTP with Netty. You may decide to handle URI's differently and may also respond with different status codes, depending on if a resource is present or not but the concept is the same.

But wait this chapter is about SPDY, isn't it? So while it's nice to be able to handle HTTP as fallback it's still not what we want as the preferred protocol. Thanks to the provided SPDY handlers by Netty this is a very easy.

Netty will just allow you to re-use the `FullHttpRequest` / `FullHttpResponse` messages and transparently receive/send them via SPDY. Thus we basically can reuse our previous written handler.

One thing to change is the content to write back to the client. This is mainly done to show the difference; in real-world you may just write back the same content.

Listing 12.3 shows the implementation, which just extends the previous written `HttipRequestHandler`.

Listing 12.3 Implementation which handles SPDY

```
public class SpdyRequestHandler extends HttpRequestHandler {            #1
    @Override
    protected String getContent() {
        return "This content is transmitted via SPDY\r\n";               #2
    }
}
```

#1 Extends HttpRequestHandler and so share the same logic
#2 Generate the content which is written to the payload. This overrides the implementation of this method in HttpRequestHandler.

The inner working of the `SpdyRequestHandler` is the same as `HttpReuestHandler`, as it extends it and so share the same logic. Only with one difference; The Content, which is written to the payload, contains the details that the response was written over SPDY.

Now with the two handlers in place it's time to implement the logic, which will choose the right one depending on what protocol is detected. But adding the previous written handler to the `ChannelPipeline` is not enough, as the correct codecs needs to also be added. Its responsibility is to detect the transmitted bytes and finally allow working with the `FullHttpResponse` and `FullHttpRequest` abstraction.

Fortunately this easier than it sounds, as Netty ships with a base class which does exactly this. All you need to do is to implement the logic to select the Protocol and which handler to use for HTTP and which for SPDY.

Listing 12.4 shows the implementation, which makes use of the abstract base class provided by Netty.

Listing 12.4 Implementation which handles SPDY

```
public class DefaultSpdyOrHttpChooser extends SpdyOrHttpChooser {

    public DefaultSpdyOrHttpChooser(int maxSpdyContentLength,
int maxHttpContentLength) {
        super(maxSpdyContentLength, maxHttpContentLength);
    }

    @Override
    protected SelectedProtocol getProtocol(SSLEngine engine) {
        DefaultServerProvider provider =
(DefaultServerProvider) NextProtoNego.get(engine);       #1
        String protocol = provider.getSelectedProtocol();
        if (protocol == null) {
            return SelectedProtocol.UNKNOWN;                             #2
        }
```

```
            switch (protocol) {
                case "spdy/2":
                    return SelectedProtocol.SPDY_2;                          #3
                case "spdy/3":
                    return SelectedProtocol.SPDY_3;                          #4
                case "http/1.1":
                    return SelectedProtocol.HTTP_1_1;                        #5
                default:
                    return SelectedProtocol.UNKNOWN;                         #6
            }
        }

        @Override
        protected ChannelInboundHandler createHttpRequestHandlerForHttp() {
            return new HttpRequestHandler();                                 #7
        }

        @Override
        protected ChannelInboundHandler createHttpRequestHandlerForSpdy() {
            return new SpdyRequestHandler();                                 #8
        }
    }
```

#1 Use he NextProtoNego to obtain a reference to the DefaultServerProvider which is in use for the SslEngine
#2 The protocol could not be detected. Once more bytes are ready to read the detect process will start again
#3 SPDY Draft 2 was detected
#4 SPDY Draft 3 was detected
#5 HTTP 1.1 was detected
#6 No known protocol detected
#7 Will be called to add the handler which will handle FullHttpRequest messages. This method is only called if no SPDY is supported by the client and so HTTPS should be used
#8 Will be called to add the handler which will handle FullHttpRequest messages. This method is called if SPDY is supported.

The implementation shown in 12.4 will take care of detecting the correct protocol and setting up the `ChannelPipeline` for you. It only handles SPDY 2 / 3 and HTTP 1.1 but could be adjusted to only support a specific version of SPDY etc.

12.3.3 Setuping the ChannelPipeline

With this we have everything in place and just need to wire the handlers together. This is done by an implementation of `ChannelInitializer` as usual. As we learned before the responsible for implementations of this is to setup the `ChannelPipeline` and so add all needed `ChannelHandlers` into it.

Because of the nature of SPDY those are actual two:

1. The `SslHandler`, as if you remember the detection of SPDY is done via a TLS Extension.
2. Our `DefaultSpdyOrHttpChooser`, which will add the correct `ChannelHandlers` into the `ChannelPipeline` once the protocol, was detected.

Besides adding the `ChannelHandlers` to the ChannelPipeline, it has also another responsibility. It assign the previous created `DefaultServerProvider` to the `SslEngine` which is used by the SslHandler in the ChannelPipeline. This is done by making use of the `NextProtoNego` helper class that is provided by the jetty NPN library.

Listing 12.5 shows the implementation in detail.

Listing 12.5 Implementation which handles SPDY

```
public class SpdyChannelInitializer extends
ChannelInitializer<SocketChannel> {                                  #1

    private final SSLContext context;

    public SpdyChannelInitializer(SSLContext context) {                  #2
        this.context = context;
    }

    @Override
    protected void initChannel(SocketChannel ch) throws Exception {
        ChannelPipeline pipeline = ch.pipeline();
        SSLEngine engine = context.createSSLEngine();                    #3
        engine.setUseClientMode(false);                                  #4

        NextProtoNego.put(engine, new DefaultServerProvider());          #5
        NextProtoNego.debug = true;

        pipeline.addLast("sslHandler", new SslHandler(engine));          #6
        pipeline.addLast("chooser",
new DefaultSpdyOrHttpChooser(1024 * 1024, 1024 * 1024));#7
    }
}}
```

#1 Extend ChannelIntializer for an easy starting point
#2 Pass the SSLContext in which will be used to create the SSLEngines from
#3 Create a new SSLEngine which will be used for the new Channel / Connection
#4 Configure the SSLEngine for non-client usage
#5 Bind the DefaultServerProvider to the SSLEngine via the NextProtoNego helper class
#6 Add the SslHandler into the ChannelPipeline, this will stay in the ChannelPipeline even after the protocol was detected.
#7 Add the DefaultSpyOrHttpChooser into the ChannelPipeline. This implementation will detect the protocol, add the correct ChannelHandlers in the ChannelPipeline and remove itself.

The "real" `ChannelPipeline` setup is done later by the `DefaultSpdyOrHttpChooser` implementation, as it is only possible to know at this point if the client supports SPDY or not.

To illustrate this let us recap and have a look at the different `ChannelPipeline` "states" during the "life-time" of the connection with the client.

Figure 12.2 shows the `ChannelPipeline`, which will be used after the `Channel` is initialized.

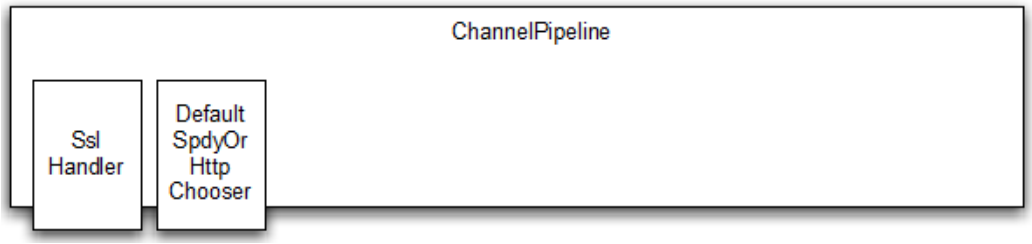


Figure 12.2 ChannelPipeline after connect

Now depending on if the client supports SPDY or not it will get modified by the `DefaultSpdyOrHttpChooser` to handle the protocol. The `DefaultSpdyOrHttpChooser` does this by first adding the required `ChannelHandlers` to the `ChannelPipeline` and then removing itself from the `ChannelPipeline` (as it is no longer needed). All this logic is encapsulated and hidden by the abstract `SpdyOrHttpChooser` class, from which `DefaultSpdyOrHttpChooser` extend.

Figure 12.3 shows the `ChannelPipeline`, which will be used if SPDY is usable for the connected client.

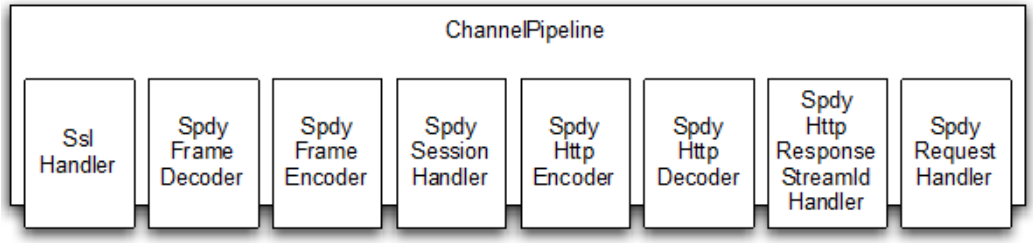


Figure 12.3 ChannelPipeline if SPDY is supported

All of the added `ChannelHandlers` have a responsibility and fulfil a small part of the work. This match perfectly the way you normally write Netty based Applications. Every "piece" does a small amount of work making it re-usable and more generic.

The responsibilities of the `Channelhandlers` are shown in Table 12.1.

Table 12.1 Responsibilities of the ChannelHandlers

Name	Responsibility
`SslHandler`	Encrypt / Decrypt data which is exchanged between the two peers
`SpdyFrameDecoder`	Decode the received bytes to SPDY frames

`SpdyFrameEncoder`	Encoder SPDY frames back to bytes
`SpdySessionHandler`	SPDY session handling
`SpdyHttpEncoder`	Encoder HTTP messages to SPDY frames
`SpdyHttpDecoder`	Decoder SDPY frames into Http messages
`SpdyHttpResponseStreamIdHandler`	Handle the mapping between requests and responses based on the SPDY id
`SpdyRequestHandler`	Handle the FullHttpRequests, which were decoded from the SPDY frame and so allow transparent usage of SPDY.

Like said before the `ChannelPipeline` will look different for when HTTP(s) is in use as the protocol, which is exchanged differs. Figure 13.4 shows the `ChannelPipeline` in this case.

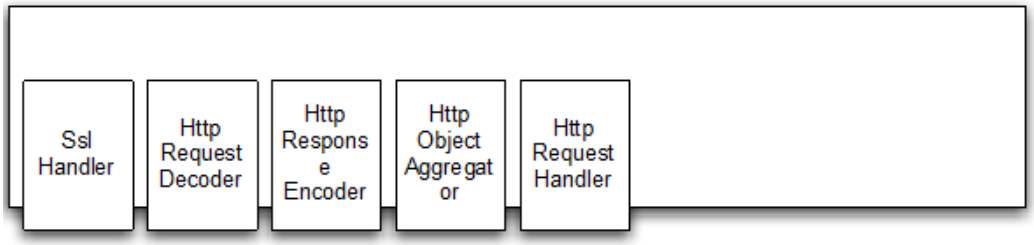


Figure 12.4 ChannelPipeline if SPDY is not supported

As before each of the `ChannelHandlers` has a responsibility, which is highlighted in Table 12.2.

Table 12.2 Responsibilities of the ChannelHandlers

Name	Responsibility
`SslHandler`	Encrypt / Decrypt data which is exchanged between the two peers
`HttpRequestDecoder`	Decode the received bytes to Http requests
`HttpResponseEncoder`	Encode Http responses to bytes
`HttpObjectAggregator`	SPDY session handling
`HttpRequestHandler`	Handle the FullHttpRequests, which were decoded

12.3.4 Wiring things together

After all the `ChannelHandler` implementations are prepared the only part, which is missing is the implementation that wires all the previous written implementation together and bootstrap the Server.

For this purpose the **`SpdyServer`** implementation is provided as shown in Listing 12.6.

Listing 12.6 SpdyServer implementation

```
public class SpdyServer {

    private final NioEventLoopGroup group = new NioEventLoopGroup();     #1
    private final SSLContext context;
    private Channel channel;

    public SpdyServer(SSLContext context) {                              #2
        this.context = context;
    }

    public ChannelFuture start(InetSocketAddress address) {
        ServerBootstrap bootstrap  = new ServerBootstrap();              #3
        bootstrap.group(group)
                .channel(NioServerSocketChannel.class)
                .childHandler(new SpdyChannelInitializer(context));      #4
        ChannelFuture future = bootstrap.bind(address);                  #5
        future.syncUninterruptibly();
        channel = future.channel();
        return future;
    }

    public void destroy() {                                              #6
        if (channel != null) {
            channel.close();
        }
        group.shutdownGracefully();
    }

    public static void main(String[] args) {
        if (args.length != 1) {
            System.err.println("Please give port as argument");
            System.exit(1);
        }
        int port = Integer.parseInt(args[0]);

        SSLContext context = BogusSslContextFactory.getServerContext(); #7

        final SpdyServer endpoint = new SpdyServer(context);
        ChannelFuture future = endpoint.start(new InetSocketAddress(port));

        Runtime.getRuntime().addShutdownHook(new Thread() {
            @Override
            public void run() {
                endpoint.destroy();
            }
```

```
            });
            future.channel().closeFuture().syncUninterruptibly();
        }
    }
```

#1 Construct a new NioEventLoopGroup which handles the IO
#2 Pass in the SSLContext to use for encryption
#3 Create a new ServerBootstrap to configure the server
#4 Configure the ServerBootstrap
#5 Bind the server so it accept connections
#6 Destroy the server which means it close the channel and shutdown the NioEventLoopGroup
#7 Obtain a SSLContext from BogusSslContextFactory. This is a dummy implementation which can be used for testing purposes. For a real implementation you would configure the SslContext to use via a proper KeyStore. Refer to the java api documentation for details.

So to be understand what it does let us have a look about the "flow":

1. The `ServerBootstrap` is created and configured with the previous created `SpdyChannelInitializer`. The `SpdyChannelInitializer` sets up the `ChannelPipeline` once the Channel was accepted
2. Bind to the given `InetSocketAddress` and so accept connections on it.

Beside it also register a shutdown hook to release all resource once the JVM exists.
Now it's time to actual test the `SpdyServer` and see if it works like expected.

12.4 Start the SpdyServer and test it

One thing that is important when you work with the jetty NPN library is that you need to pass a special argument to the JVM when starting it. Otherwise it will not work, this is needed because of how it hooks into the **`SslEngine`**. So while we won't go into details how you compile the code here be aware that once it is compiled and you start it via the java command you will need to take special care here ensuring you set the "-Xbootclasspath" JVM argument to the path of the npn boot JAR file. The "-Xbootclasspath" allows to override standard implementations of classes that are shipped with the JDK itself.

Listing 12.7 shows the special argument (-Xbootclasspath) to use.

Listing 12.7 SpdyServer implementation

```
java -Xbootclasspath/p:<path_to_npn_boot_jar> ….
```

The easiest way to start up the application is to use the source-code provided for this book. It has everything ready to use and will use maven automatically to setup everything and download dependencies. Starting it up via mvn is as easy as shown in Listing 12.8

Listing 12.8 Compile and start SpdyServer with maven

```
Normans-MacBook-Pro:netty-in-action-privatenorman$ mvn clean package
exec:exec -Pchapter12-SpdyServer
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------
-----------
```

```
    [INFO] Building netty-in-action 0.1-SNAPSHOT
    [INFO] ------------------------------------------------------------------
    ----------
    ...
    ...
    ...
    [INFO]
    [INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ netty-in-action
    ---
    [INFO] Building jar: /Users/norman/Documents/workspace-
    intellij/netty-in-action-private/target/netty-in-action-0.1-
    SNAPSHOT.jar
    [INFO]
[INFO] --- exec-maven-plugin:1.2.1:exec (default-cli) @ netty-in-action
---
```

Once the Server is successfully started it's time to actual test it. For this purpose we will use two different Browsers. One which supports SPDY and one which does not. At the time of writing this is Google Chrome (supports SPDY) and Safari (not supports SPDY). But it would also be possible to use Google Chrome and for example a unix tool like `wget` to test for fallback to HTTP(s).You may get a "warning" if you use a self-signed certificate, but that's ok. For a production system you probability want to buy an official certificate.

Open up Google Chrome and navigate to the url "https://127.0.0.1:9999" or the matching ipaddress / port that you used.

Figure 12.4 shows it in action.

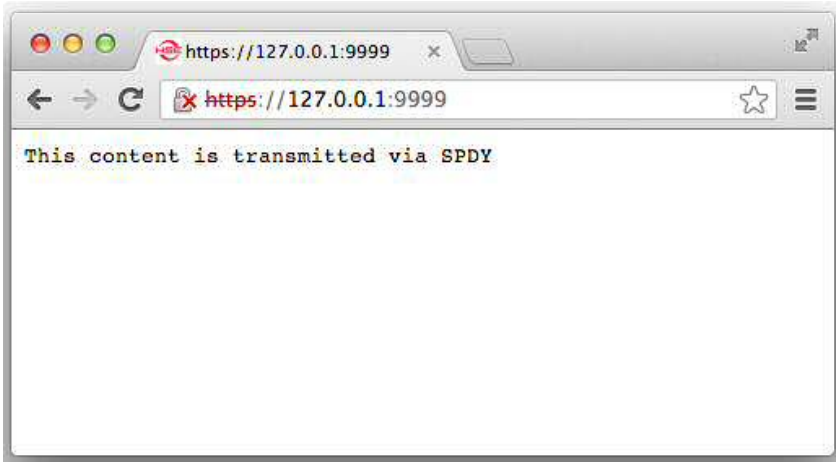


Figure 12.5 SPDY supported by Google Chrome

If you look at Figure 12.4 you should see that it's served via SPDY, as it use the previous created `SpdyRequestDecoder` and so writes "This content is transmitted via SPDY" back to the client.

A nice feature in Google Chrome is that you can check some statistics about SPDY with it and so get a deeper understanding what is transmitted and how many connections are currently handled by SPDY. For this purpose open the url "chrome://net-internals/#spdy".

Figure 12.5 shows the statistics, which are exposed here.

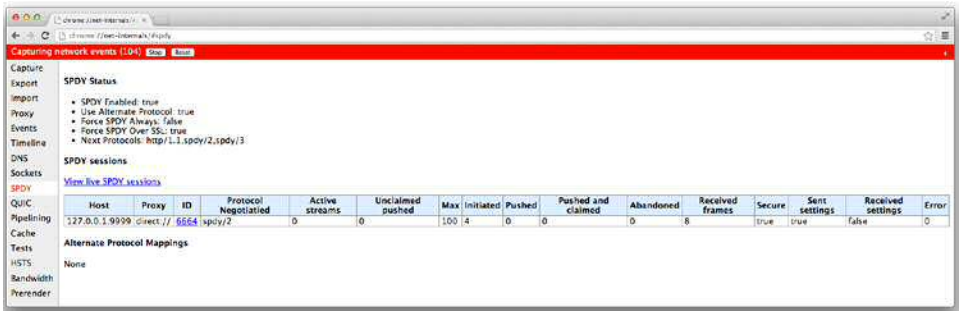


Figure 12.6 SPDY statistics

This gives you an overview over all the SPDY connections, which are established. And some more details about how many bytes are transmitted and many more. To get an even more detailed overview about the transmitted data click on the "ID" link and refresh the window in which you haven opened "https://127.0.0.1:9999". This will give you details about the different SPDY frames which are transmitted to fulfill the request.

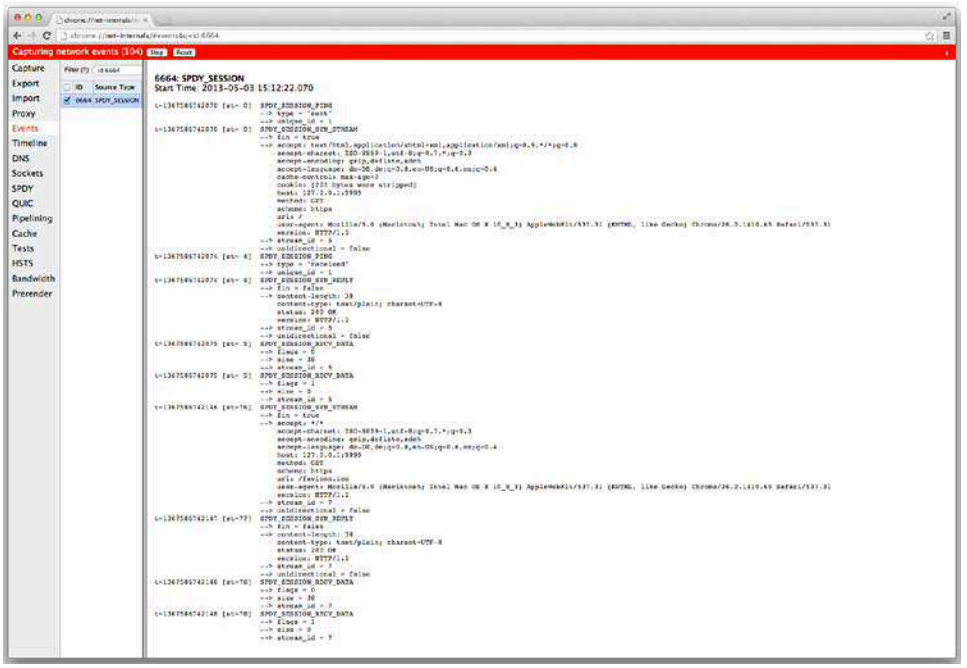


Figure 12.7 SPDY statistics

Going through all the details displayed in Figure 12.6 is out of scope of this book, but you should be able to spot easily some details about which HTTP headers are transmitted and which url was accessed. These kinds of statistics are quite useful if you debug SPDY in general.

What about the clients that do not support SPDY? Those will just be served via HTTPS and so should not have a problem to access the resources provided by the `SpdyServer`.

To test this open up a browser,which does not support SPDY, which is Safari in our case. Like before access the url "https://127.0.0.1" and see what response is written back the client (browser).

Figure 12.7 shows it.

Figure 12.8 SPDY not supported by Safari

Figure 12.7 makes it clear that it's served via HTTP(s) as it writes back "This content is transmitted via HTTP" to the client. This is part of the `HttpRequestHandler` and so should be quite familiar to you.

12.5 Summary

In this chapter you learned how easy it is to use SPDY and HTTP(s) at the same time in a Netty based application. The shown application should give you a good starting point to support new clients (with support of SPDY) and the best performance while also allowing "old" clients to access the application.

You learned how you make use of the provided helper classes for SPDY in Netty and so re-use what is provided out of the box. Also you saw how you can use Google Chrome to get more information about SPDY connections and so understand what happens under the hood.

The shown techniques should help you and adapt them in your next Netty based application. Also you saw again how modification of the `ChannelPipeline` can help you to build powerful "multiplexers" which allow you to switch between different protocols during one connection.

In the next chapter you will learn how you can use UDP to write apps and make use of its high-performance connection-less nature.

13 Broadcasting events via UDP

In this Chapter

- UDP in general
- ChannelHandler, Decoder and Encoder
- Bootstrap an Netty based Application

The previous chapters have focused on examples which use connection based protocols such as TCP. In this chapter we will focus on using UDP. UDP is a connection-less protocol, which is normally used if performance is most critical and loosing packets is not an issue. One of the most well known UDP based protocols is DNS, which is used for domain name resolution.

Thanks to its unified Transport API, Netty makes writing UDP based applications easy as taking a breath. This allows you to reuse existing `ChannelHandlers` and other utilities that you have written for other Netty based Applications.

After this chapter you will know what is special about Connection-less Protocols, in this case UDP. And why UDP may be a good fit for your Application.

This is a self contained chapter. You will be able to go through the chapter even without reading other chapters of the book first. It should contain everything you need to get the example application running an gain an understanding of UDP in Netty. While the chapter is self contained, we won't re-iterate previous topics in great detail. Knowing the basics of the Netty API (as covered in earlier chapters) is more than sufficient for you to get through.

13.1 Handle UDP

Before we dive into the application the rest of this chapter focuses on, we're take a moment to look at UDP, what it is and the various limitations (or at least issues) that you should be aware of before using it.

As stated before UDP is connection-less. Which means that there is not "connection" for the lifetime of the communication between clients and servers. So it is not so easy to say all those messages belong to "one connection".

Another "limitation" is that UDP has no "error correction" like TCP. In TCP if packets are lost during transmission the peer that sends it will know because the receiver sends an acknowledgement of packets received. Failure to receive an acknowledgement causes the peer to re-sent the same packets again. This is not true for UDP. UDP is more like "fire and forget", which means it has no idea if the data sent was received at all.

Those "limitations" are one of the reasons why UDP is so fast compared to TCP. As it eliminates all the overhead of handshaking and other overhead that comes with TCP.

Knowing this should hopefully point out to you that that UDP is only a good fit for applications that can "handle" the case of lost messages. Which means it is not a very good fit for applications that can't lose any message, like an application that handles money transactions. In the next section we're going to design the UDP application we use for the rest of this chapter.

13.2 UDP Application structure & Design

The rest of this chapter will write an application, which will broadcast messages via UDP to another part, which will receive them. In this case the application opens up a file and broadcasts every new line via UDP.

If you are familiar with UNIX-like operation systems this may sound very familiar to you because this is what Syslog does. In fact this is a simplified version of it.

UDP is a perfect fit for such an application as it may not be to dramatic if you loose one log line as it is stored on the base-system anyway. More importantly for the system may be the ability to send a lot of "events" over the network.

Also one additional advantage is that by using UDP broadcast there is no extra actions needed for adding new "monitors" that receive the log messages via UDP. All that is needed is to fire up the instance on the specific port and you will see the log messages flow in. But this is not only a "good thing", as everyone will be able to get the log messages once he is able to start an instance. This is one of the reasons why UDP broadcast should only be used in secure environments / networks where such things are not an issue. Also be aware that broadcast messages usually only work in the same network as routers will usually block them. These types of applications are typically classified as "pub-sub", where one side of the application or service, "publishes" events and one or more others are able to "subscribe" to these events.

Before we go any further, let's take a high-level look at the application we're going to build.

Figure 13.1 shows an overview over the application layout

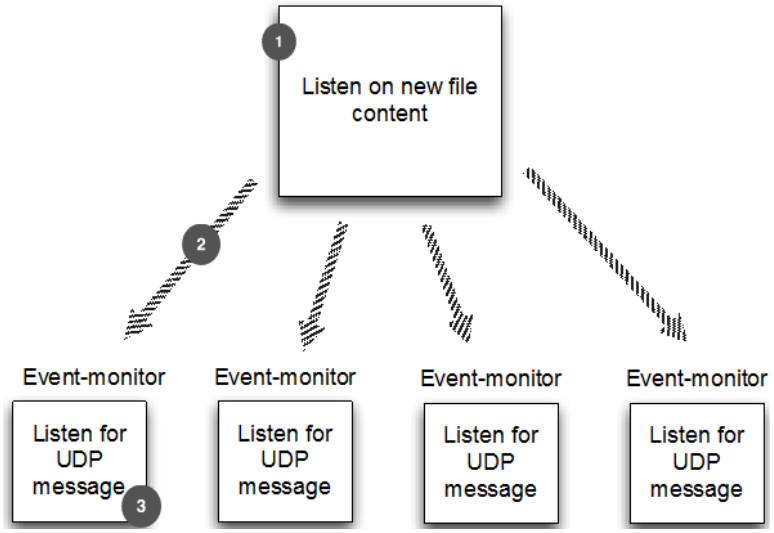


Figure 13.1 Application overview

#1 Application listen for new file content
#2 Broadcast file content via UDP
#3 Print out new log content

So the Application will consist of two parts. The log file broadcaster and the "monitor", which will receive the broadcast. Where there could be more then one receiver.

To keep things simple we will not do any kind of authentication,verification or encryption.

If you feel the application may be a good fit for one of your needs you can adjust it for your needs later by replacing parts of it. This should be quite simple as the logic is broken down into easyto follow sections.

The next section will give you some insight about what you need to know about UDP and what is the difference compared to writing TCP – protocol –based applications.

13.3 EventLog POJOs

As in other applications that are not based on Netty, you usually have some kind of "message POJO" that is used to hold application-based information to pass around. Think of it as an "Event message".

For our Application we will do this too and share it between both parts of the Application. We call it `LogEvent`, as it stores the event data, which was generated out of the Log file.

Listing 13.1 shows the simple POJO.

Listing 13.1 LogEvent message

```
public final class LogEvent {
public static final byte SEPARATOR = (byte) ':';                    #1
```

```
    private final InetSocketAddress source;
    private final String logfile;
    private final String msg;
    private final long received;

    public LogEvent(String logfile, String msg) {                    #1
        this(null, -1, logfile, msg);
    }

    public LogEvent(InetSocketAddress source, long received,
String logfile, String msg) {                                        #2
        this.source = source;
        this.logfile = logfile;
        this.msg = msg;
        this.received = received;
    }

    public InetSocketAddress getSource() {                           #3
        return source;
    }

    public String getLogfile() {                                     #4
        return logfile;
    }

    public String getMsg() {                                         #5
        return msg;
    }

    public long getReceivedTimestamp() {                             #6
        return received;
    }
}
```

#1 Instance new LogEvent for sending purposes
#2 Instance new LogEvent for receiving pruposes
#3 Get the InetSocketAddress of the source which send the LogEvent
#4 Get the name of the log file for which the LogEvent was send
#5 Get the actual log message of the LogEvent
#6 Get the timestamp at which the LogEvent was received

With the common / shared part ready it's time to implement the actual logic. In the next sections we will focus on doing this.

We will start with writing the "Broadcaster" in the next section and explain how it works exactly.

13.4 Writing the "Broadcaster"

Like illustrated before one "part" of our Application is the Log broadcaster. In this section we will go through all the steps that are needed to write it.

What we want to do is to broadcast one `DatagramPacket` per Log entry as shown in Figure 13.2.

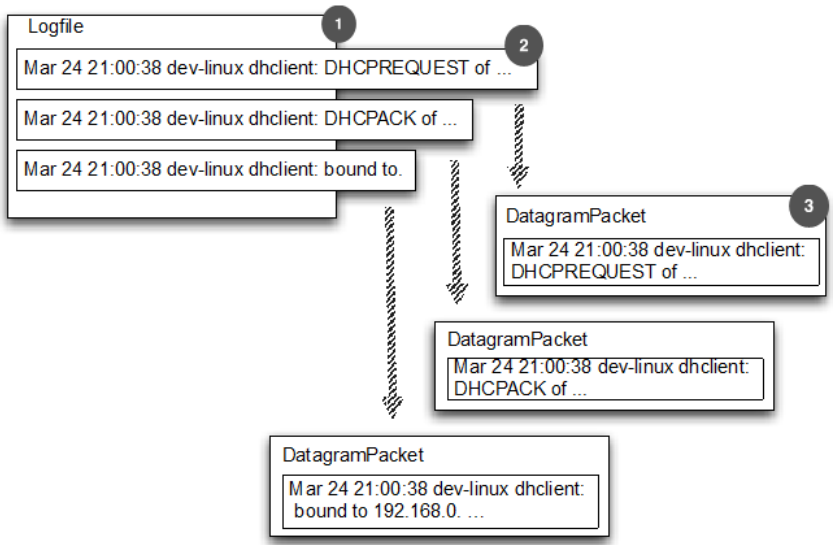


Figure 13.2 Log entry to DatagramPacket

#1 The Log file
#2 Alog entry in the Log file
#3 The DatagramPacket which holds the log entry

As shown in figure 13.2 this means we have a 1-to-1 relation from Log entries to `DatagramPackets`.

As with all Netty based applications it consist of one or more `ChannelHandler` and some entry bound that bootstrap the application.

First off let us have a look at the `ChannelPipeline` of the `LogEventBroadcaster` and how the `LogEvent` (which was created before) will flow through it. Figure 13.3 gives some high-level overview.

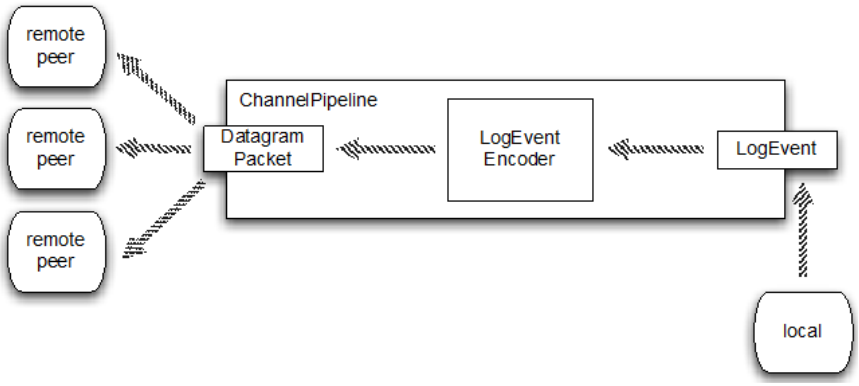


Figure 13.3LogEventBroadcasterChannelPipeline and LogEvent flow

So let's recap on Figure 13.3. The LogEventBroadcaster use LogEvent messages and write them via the Channel on the local side. This means all the information is encapsulated in the LogEvent messages. Those messages are then sent trough the ChannelPipeline.

While flowing through the `ChannelPipeline` the `LogEvent` messages are encoded to `DatagramPacket` messages, which finally are broadcasted via UDP to the remote peers.

This boils down to the need of having one custom `ChannelHandler`, which encodes from `LogEvent` messages to `DatagramPacket` messages. Remembering what we learned in Chapter 7 (Codec API) it should be quite clear what kind of abstract base class for the `LogEventEncoder` should be used to keep the amount of code to be written to the minimum and so make it easier to maintain. If you've not read Chapter 7 yet, don't worry. You will be able to go through this chapter without it.

Listing 13.2 shows how the encoder was implemented.

Listing 13.2 LogEventEncoder

```
public class LogEventEncoder extends MessageToMessageEncoder<LogEvent> {
    private final InetSocketAddress remoteAddress;

    public LogEventEncoder(InetSocketAddress remoteAddress) {           #1
        this.remoteAddress = remoteAddress;
    }

    @Override
    protected void encode(ChannelHandlerContext channelHandlerContext,
LogEvent logEvent, List<Object> out)
throws Exception {
        ByteBuf buf = channelHandlerContext.alloc().buffer();
        buf.writeBytes(logEvent.getLogfile()
.getBytes(CharsetUtil.UTF_8));#2
        buf.writeByte(LogEvent.SEPARATOR);                              #3
        buf.writeBytes(logEvent.getMsg().getBytes(CharsetUtil.UTF_8));  #4
```

```
out.add(new DatagramPacket(buf, remoteAddress));                #5
    }
}
```

#1 Construct a new LogEventEncoder which will create DatagramPacket messages that will send to the given InetSocketAddress
#2 Write the filename in the ByteBuf
#3 Separate it via the SEPARATOR
#4 Write the actual log message to the ByteBuf
#5 Add the DatagramPacket to the List of encoded messages.

Why MessageToMessageEncoder ?

MessageToMessageEncoder was used in Listing 13.2, as the implementation needs to encode one Message type to another one. MessageToMessageEncoder allows this in an easy way and so offer an easy-to-use abstract. But keep in mint MessageToMessageEncoder is just a ChannelOutboundHandler implementation, so you could also just have implement the ChannelOutboundHandler interface directly. MessageToMessageEncoder just makes it a lot easier.

After we were able to build the `LogEventEncoder` we need to bootstrap the Application and use it. Bootstrapping is the part of the application, which is responsible to configure the actual Bootstrap. This includes set various `ChannelOptions` and some `ChannelHandlers` that are used in the `ChannelPipeline` of the `Channel`.

This is done in Listing 13.3 by our `LogEventBroadcaster` class.

Listing 13.3LogEventBroadcaster

```
public class LogEventBroadcaster {
   private final EventLoopGroup group;
    private final Bootstrap bootstrap;
    private final File file;

    public LogEventBroadcaster(InetSocketAddress address,
File file) {
        group = new NioEventLoopGroup();
        bootstrap = new Bootstrap();
            bootstrap.group(group)
                    .channel(NioDatagramChannel.class)
                    .option(ChannelOption.SO_BROADCAST, true)
                    .handler(new LogEventEncoder(address));          #1

        this.file = file;
    }

    public void run() throws IOException {
        Channel ch = bootstrap.bind(0).syncUninterruptibly().channel(); #2
        long pointer = 0;
        for (;;) {
```

```
            long len = file.length();
            if (len < pointer) {
                // file was reset
                pointer = len;                                          #3
            } else if (len > pointer) {
                // Content was added
                RandomAccessFile raf = new RandomAccessFile(file, "r");
                raf.seek(pointer);                                  #4
                String line;
                while ((line = raf.readLine()) != null) {
                    ch.write(new LogEvent(null, -1,
file.getAbsolutePath(), line));             #5
                }
                pointer = raf.getFilePointer();                         #6
                raf.close();
            }
            try {
                Thread.sleep(1000);                                  #7
            } catch (InterruptedException e) {
                Thread.interrupted();
                break;
            }
        }
    }

    public void stop() {
group.shutdown();
    }

    public static void main(String[] args) throws Exception {
        if (args.length != 2) {
            throw new IllegalArgumentException();
        }

        LogEventBroadcaster broadcaster = new LogEventBroadcaster(
new InetSocketAddress("255.255.255.255",
                Integer.parseInt(args[0])), new File(args[1]));         #8
        try {
            broadcaster.run();
        } finally {
            broadcaster.stop();
        }
    }
}
```

#1 Bootstrap the NioDatagramChannel. It is important to set the SO_BROADCAST option as we want to work with broadcasts
#2 Bind the Channel so we can use it. Note that there is no connect when use DatagramChannel as those are connectionless.
#3 Set the current filepointer to the last byte of the file as we only want to broadcast new log entries and not what was written before the application was started
#4 Set the current filepointer so nothing old is send
#5 Write a LogEvent to the Channel which holds the filename and the log entry (we exepect every logentry is only one line long)
#6 Store the current position within the file so we can later continue here

```
#7 Sleep for 1 second and then check if there is something new in the logfile which we can send
#8 Construct a new instance of LogEventBroadcaster and start it.
```

This is the point where we have the first part of our Application done. We can see the first results at this point, event without the second part. For this let us fire up the netcat program, which should be installed if you use a UNIX-like OS or can be installed on windows via the available installer on the internet.

Netcat allows you to listen on a specific port and just print all data received to STDOUT. This is perfect to test-drive our implementation.

Listing 13.4 shows how you can start Netcat to listen for UDP data on port 9090.

Listing 13.4Netcat in action

```
normans-macbook-pro: norman$ nc -l -u 9090
```

After this is done it's time to startup `LogEventBroadcaster` pointing it to a log file that changes in short intervals and using port 9090.

Once the log file gets a new entry written it will broadcast the entry via UDP, and so it will be written out in the console in which you have started the `nc` command before.

Using Netcat for testing purposes is an easy thing, but not a production System. This brings us the second part of the Application, which we will implement in the next section of this Chapter.

13.5 Writing the "monitor"

Having the first part of the application in place it's time to replace our Netcat usage with some real code. For this we will develop the `EventLogMonitor` program in this section.

This program will do the following:

- Receive UDP DatagramPackets that was broadcasted via the LogEventBroadcaster
- Decode them to LogEvent messages
- Handle the LogEvent messages by write them to System.out

To handle the logic we will write our own ChannelHandler implementations again. Let us have a look on the `ChannelPipeline` of the `LogEventMonitor` and how the `LogEvent` (which was created before) will flow through it.

Figure 13.4 shows the details.

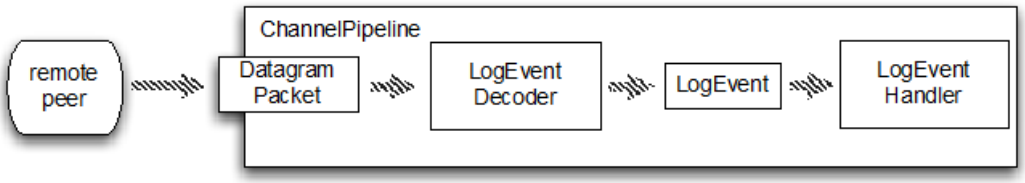


Figure 13.4LogEventMontorChannelPipeline and LogEvent flow

In Figure 13.4 you see there are two custom `ChannelHandler` implementations present. Let us give a spin on implement both of them.

The first one to implement is the `LogEventDecoder`. The `LogEventDecoder` is responsible to decode `DatagramPackets` that are received over the network into `LogEvent` messages. Again this is how most of your Netty applications will start if you receive data on which you operate.

Listing 13.4 shows how the encoder implementation looks like.

Listing 13.4 LogEventDecoder

```
public class LogEventDecoder extends MessageToMessageDecoder<DatagramPacket>
{

    @Override
    protected void decode(ChannelHandlerContext channelHandlerContext,
DatagramPacket datagramPacket, List<Object> out)
throws Exception {
        ByteBuf data = datagramPacket.data();                          #1
        int i = data.indexOf(0, data.readableBytes(),
LogEvent.SEPARATOR);                                        #2
        String filename = data.slice(0, i)
.toString(CharsetUtil.UTF_8);                              #3
        String logMsg =  data.slice(i + 1,
data.readableBytes()).toString(CharsetUtil.UTF_8);         #4

        LogEvent event = new LogEvent(datagramPacket.remoteAddress(),
System.currentTimeMillis(),filename,logMsg);               #5
out.add(event);
    }
}
```

#1 Get a reference to the data in the DatagramPacket
#2 Get the index of the SEPARATOR
#3 Read the filename out of the data
#4 Read the log message out of the data
#5 Construct a new LogEvent object and finally return it

Having the decoder in place it is time to write the actual handler that will do something with the `LogEvent` messages. In our case we just want them to get written to System.out. In a real-world application you may want to write them in some aggregated logfile, database or do something useful with it. This is just to show you the moving parts.

Listing 13.5shows the LogEventHandler that writes the content of the LogEvent to System.out.

Listing 13.4 LogEventHandler

```
public class LogEventHandler
extends SimpleChannelInboundHandler<LogEvent> {         #1

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

```
            cause.printStackTrace();                                    #2
            ctx.close();
        }

        @Override
        public void channelRead0(ChannelHandlerContext ctx,
    LogEvent event) throws Exception {
            StringBuilder builder = new StringBuilder();                #3
            builder.append(event.getReceivedTimestamp());
            builder.append(" [");
            builder.append(event.getSource().toString());
            builder.append("] [");
            builder.append(event.getLogfile());
            builder.append("] : ");
            builder.append(event.getMsg());

            System.out.println(builder.toString());                     #4
        }
    }
```

#1 Extend SimpleChannelInboundHandler to handle LogEvent messages
#2 Print the Stacktrace on error and close the channel
#3 Create a new StringBuilder and append the infos of the LogEvent to it
#4 Print out the LogEvent data

The LogEventHandler prints the `LogEvents`out in an easy to read format which consist out of those:

- Received timestamp in milliseconds
- `InetSocketAddress` of the sender which consist of ip address and port
- The absolute name of the file which the `LogEvent` was generated from
- The actual log message, which represent on line in the log file.

Having now written the decoder and the handler, it's time to assemble the parts to the previously outlined `ChannelPipeline`.

Listing 13.5 shows how this is done as part of the `LogEventMonitor` class.

Listing 13.5 LogEventMonitor

```
public class LogEventMonitor {
    private final EventLoopGroup group;
    private final Bootstrap bootstrap;

    public LogEventMonitor(InetSocketAddress address) {
        group = new NioEventLoopGroup();
        bootstrap = new Bootstrap();
        bootstrap.group(group)
                .channel(NioDatagramChannel.class)
                .option(ChannelOption.SO_BROADCAST, true)
                .handler(new ChannelInitializer<Channel>() {
                    @Override
                    protected void initChannel(Channel channel)
throws Exception {
```

```
                        ChannelPipeline pipeline = channel.pipeline();
                        pipeline.addLast(new LogEventDecoder());
                        pipeline.addLast(new LogEventHandler());
                    }
                }).localAddress(address);                                    #1

    }

    public Channel bind() {
        return bootstrap.bind().syncUninterruptibly().channel();             #2
    }

    public void stop() {
group.shutdown();
    }

    public static void main(String[] main) throws Exception {
        if (args.length != 1) {
            throw new IllegalArgumentException(
"Usage: LogEventMonitor <port>");
        }
        LogEventMonitor monitor = new LogEventMonitor(
new InetSocketAddress(args[0]));                                     #3
try {
            Channel channel = monitor.bind();
            System.out.println("LogEventMonitor running");

            channel.closeFuture().await();
        } finally {
            monitor.stop();
        }
}
}
```

#1 Bootstrap the NioDatagramChannel. It is important to set the SO_BROADCAST option as we want to work with broadcasts
#2 Bind the Channel so we can use it. Note that there is no connect when use DatagramChannel as those are connectionless.
#3 Constrcut a new LogEventMonitor and use it.

13.6 Using the LogEventBroadcaster and LogEventMonitor

Having written our Application its time to see it in action. The easiest way is again to use maven. Please see Chapter 2 if you need some guidance how to setup maven and the other parts to run the code. I will not go into details here.

You will need to open up two different console windows as both of the Applications will keep running until you stop them via "CRTL+C".

First off let us start LogEventBroadcaster, which will broadcast the actual log messages via UDP. Listing 13.5 shows how this can be done using the mvn command.

Listing 13.5 Compile and start the LogEventBroadcaster

```
Normans-MacBook-Pro:netty-in-action-private norman$ mvn clean package
exec:exec -Pchapter13-LogEventBroadcaster
```

```
    [INFO] Scanning for projects...
    [INFO]
    [INFO] ------------------------------------------------------------------------
    ---
    [INFO] Building netty-in-action 0.1-SNAPSHOT
    [INFO] ------------------------------------------------------------------------
    ---
    ...
    ...
    ...
    [INFO]
    [INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ netty-in-action ---
    [INFO] Building jar: /Users/norman/Documents/workspace-intellij/netty-in-
    action-private/target/netty-in-action-0.1-SNAPSHOT.jar
    [INFO]
[INFO] --- exec-maven-plugin:1.2.1:exec (default-cli) @ netty-in-action -
-

    LogEventBroadcaster running
```

By default it will use "/var/log/messages" as log file and port 9999 to send the log messages to. If you want to change this you can do it with the logfile and port System properties when starting the LogEventBroadcaster.

Listing 13.6 shows how you can use port 8888 and the log file "/var/log/mail.log".

Listing 13.6 Compile and start the LogEventBroadcaster

```
    Normans-MacBook-Pro:netty-in-action-private norman$ mvn clean package
    exec:exec -Pchapter13-LogEventBroadcaster -Dport=8888 -
    Dlogfile=/var/log/mail.log
    ....
    ....
    [INFO]
[INFO] --- exec-maven-plugin:1.2.1:exec (default-cli) @ netty-in-action -
-

    LogEventBroadcaster running
```

When you see "LogEventBroadcaster running" you know it started up successfully. If any error was thrown it will print out an Exception.

Having the `LogEventBroadcaster` running it will start to write out log messages once those are written to the log file.

Get the Log messages sent via UDP is nice but without doing something with them its kind of useless. So time for us to start up the `LogEventMonitor` to receive and handle them. Again we will use maven for doing so as everything is prepared for it.

Listing 13.7 Compile and start the LogEventBroadcaster

```
    Normans-MacBook-Pro:netty-in-action-private norman$ mvn clean package
exec:exec -Pchapter13-LogEventMonitor
```

```
    [INFO] Scanning for projects...
    [INFO]
    [INFO]  ------------------------------------------------------------------------
---
    [INFO] Building netty-in-action 0.1-SNAPSHOT
    [INFO]  ------------------------------------------------------------------------
---
    ...
    [INFO]
    [INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ netty-in-action ---
    [INFO]  Building  jar:  /Users/norman/Documents/workspace-intellij/netty-in-
action-private/target/netty-in-action-0.1-SNAPSHOT.jar
    [INFO]
    [INFO] --- exec-maven-plugin:1.2.1:exec (default-cli) @ netty-in-action ---
    LogEventMonitor running
```

When you see "LogEventMonitor running" you know it started up successfully. If any error was thrown it will print out an Exception. Now watch the console and see how new log messages will start to get printed in it once they are written to the log file.

Listing 13.8 shows how such an output will look like.

Listing 13.8LogEventMonitor output

```
    1364217299382 [/192.168.0.38:63182] [/var/log/messages] : Mar 25 13:55:08
    dev-linux dhclient: DHCPREQUEST of 192.168.0.50 on eth2 to 192.168.0.254
    port 67141414141414141414141414141414141414141414141414
    1364217299382 [/192.168.0.38:63182] [/var/log/messages] : Mar 25 13:55:08
    dev-linux dhclient: DHCPACK of 192.168.0.50 from
    192.168.0.25414141414141414141414141414141414141414141414
    1364217299382 [/192.168.0.38:63182] [/var/log/messages] : Mar 25 13:55:08
    dev-linux dhclient: bound to 192.168.0.50 -- renewal in 270
    seconds.1414141414141414141414141414141414141414141414
    1364217299382 [/192.168.0.38:63182] [[/var/log/messages] : Mar 25 13:59:38
    dev-linux dhclient: DHCPREQUEST of 192.168.0.50 on eth2 to 192.168.0.254
    port 67141414141414141414141414141414141414141414141414
    1364217299382 [/192.168.0.38:63182] [/[/var/log/messages] : Mar 25 13:59:38
    dev-linux dhclient: DHCPACK of 192.168.0.50 from
    192.168.0.25414141414141414141414141414141414141414141414
    1364217299382 [/192.168.0.38:63182] [/var/log/messages] : Mar 25 13:59:38
    dev-linux dhclient: bound to 192.168.0.50 -- renewal in 259
    seconds.1414141414141414141414141414141414141414141414
    1364217299383 [/192.168.0.38:63182] [/var/log/messages] : Mar 25 14:03:57
    dev-linux dhclient: DHCPREQUEST of 192.168.0.50 on eth2 to 192.168.0.254
    port 67141414141414141414141414141414141414141414141414
    1364217299383 [/192.168.0.38:63182] [/var/log/messages] : Mar 25 14:03:57
    dev-linux dhclient: DHCPACK of 192.168.0.50 from
    192.168.0.25414141414141414141414141414141414141414141414
```

```
1364217299383 [/192.168.0.38:63182] [/var/log/messages] : Mar 25 14:03:57
dev-linux dhclient: bound to 192.168.0.50 -- renewal in 285
seconds.151515151515151515151515151515151515151515151515
```

If you look at the output in the console you will notice it contains all the information of the `LogEvent` in the format like specified in the `LogEventHandler`.

If you have no "real" log file to use you can also create a custom file, which will fill content in by hand to see our Application in action.

For doing so we use UNIX commands. If you are using Windows you can just create an empty file with notepad editor. In Listing 13.10 we first use touch to create an empty log file which we want to broadcast the `LogEvents` for.

Listing 13.10 Create logfile for testing

```
Normans-MacBook-Pro:netty-in-action-private norman$ touch ~/mylog.log
```

Now having the log file in place startup the `LogEventBroadcaster` again and point it to it. As you may remember you can set it via the logile System property.

Listing 13.11 shows it.

Listing 13.11 Use created logfile

```
Normans-MacBook-Pro:netty-in-action-private norman$ mvn clean package
exec:exec -Pchapter13-LogEventBroadcaster -Dport=8888 -Dlogfile=~/mylog.log
....
```

Once the `LogEventBroadcaster` is running again you can start to fill the log file with messages and so see how they are broadcasted. Doing this is as easy as just using `echo` and directing the output to the file.

Listing 13.12 shows how this is done.

Listing 13.12 Fill logfile with data

```
Normans-MacBook-Pro:netty-in-action-private norman$ echo 'Test log entry' >>
~/mylog.log
```

You can repeat the `echo` as often as you like and see its input written in the console in which you run the `LogEventMonitor`.

One thing to keep in mind again is that you can start as many instances of the monitor as you like and watch all of them receive the same messages without the broadcaster application changing at all.

13.7 Summary

In this chapter you learned how you will be able to use Netty to write UDP based Applications while still using the same API as when you use TCP as underlying Transport protocol. You also saw how you usually divide logic into different `ChannelHandler` and add them to the

`ChannelPipeline`. We did this by separating the decoder logic from the actual logic that handles the message object.

You got an introduction what is special about connection-less protocols like UDP and what is needed to know to work with them and use them in your next application.

After this last example it is time to have a deeper look in the more advanced features and details of Netty. This will be part of the next section of the book and span several chapters.

14 Implement a custom codec

This chapter will show you how you can easily implement custom codecs with Netty for your favorite protocol. Those codecs are easy to reuse and test, thanks to the flexible architecture of Netty.

To make it easier to gasp, Memcached is used as example for a protocol, which should be supported via the codec.

Memcached is "free & open source, high-performance, distributed memory object caching system, generic in nature, but intended for use in speeding up dynamic web applications by alleviating database load." Memcached is in effect an in-memory key-value store for small chunks of arbitrary data.

You may ask yourself "Why Memcached"?

In short, simplicity and coverage. The Memcached protocol is very straightforward; this makes it ideal for inclusion in a single chapter, without let it become to big.

14.1 Scope of the codec

We will only implement a subset of the Memcached protocol, just enough for us to add, retrieve and remove objects. This is supported by the SET, GET and DELETE commands, which are part of the Memcached protocol.

Memcached supports many other commands, but focus on just three of them allows us to better explain what is needed while keep things straight. Everything learned here could be used to latter implement the missing commands.

Memcached has a binary and plain text protocol. Both of them can be used to communicate with a Memcached server, depending on if the server supports either one of them or all. This chapter focuses on implement the binary protocol as often users are faced with implement protocols that are binary.

14.2 Implementing The Memcached codec

Whenever you are about to implement a codec for a given protocol, you should spend some time to understand its workings. Often the protocol itself is documented in some great detail. How much detail you will find here depends. Fortunately the binary protocol of Memcached is written down in great extend.

The specification for this is written up in an RFC style document and available at https://code.google.com/p/Memcached/wiki/MemcacheBinaryProtocol. As mention before we won't implement the entire protocol in this chapter instead we're going to only implement three operations, namely SET, GET and DELETE. This is done to keep things simple and not extend the scope of the chapter. You can easily adapt the classes and so add support for other operations by yourself. The shown code serves as examples and should not be taken as a full implementation of the protocol itself.

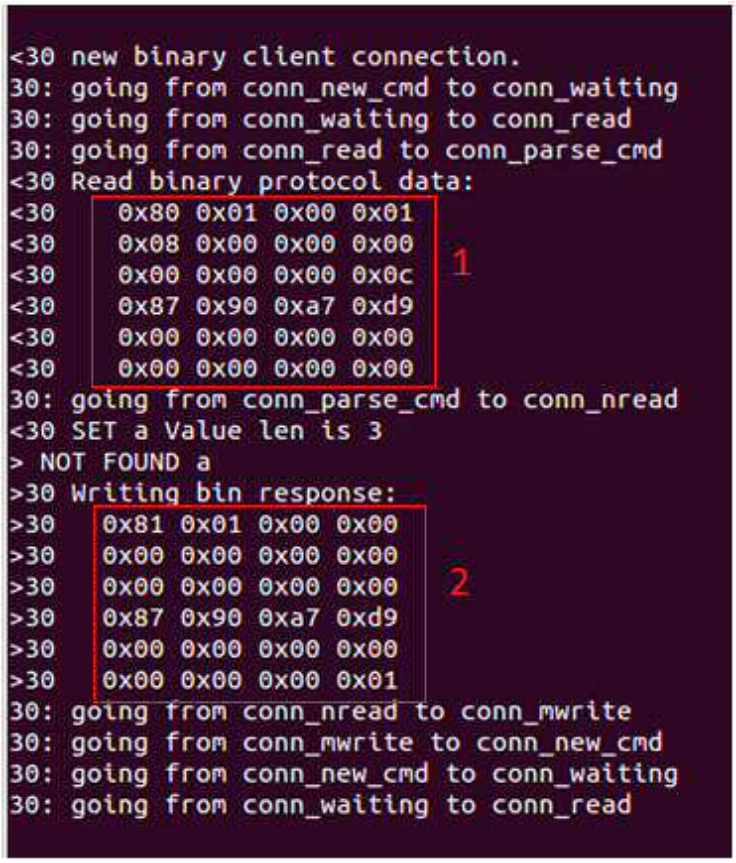
14.3 Getting to Know the Memcached Binary Protocol

So as we said, we're going to implement Memcached's GET, SET and DELETE operations (also referred to interchangeably as opcodes). We'll focus on these but Memcached's protocol has a generic structure where only a few parameters change in order to change the meaning of a request or response. This means that you can easily extend the implementation to add other commands. In general the protocol has a 24 bytes header for requests and responses. This header can be broken down in exactly the same order as denoted by the "Byte offset" column in table 14.1.

Table 14.1 Sample Memcached header byte structure

Field	Byte offset	Value
Magic	0	0x80 for requests 0x81 for responses
OpCode	1	0x01...0x1A
Key length	2 and 3	1... 32,767
Extra length	4	0x00, x04 or 0x08
Data type	5	0x00
Reserved	6 and 7	0x00
Total body length	8 - 11	Total size of body inclusive extras
Opaque	12 - 15	Any signed 32 bit integer, this one will be also included in the response and so make it easier to map requests to responses.

Notice how many bytes are used in each section. This tells us what data type we should use later. i.e. if a byte offset uses only byte 0 then we use a Java byte to represent it, if it uses 6 and 7 (2 bytes) we use a Java short, if it uses 12-15 (4 bytes) we use a Java int and so on.




```
<30 new binary client connection.
30: going from conn_new_cmd to conn_waiting
30: going from conn_waiting to conn_read
30: going from conn_read to conn_parse_cmd
<30 Read binary protocol data:
<30    0x80 0x01 0x00 0x01
<30    0x08 0x00 0x00 0x00
<30    0x00 0x00 0x00 0x0c
<30    0x87 0x90 0xa7 0xd9
<30    0x00 0x00 0x00 0x00
<30    0x00 0x00 0x00 0x00
30: going from conn_parse_cmd to conn_nread
<30 SET a Value len is 3
> NOT FOUND a
>30 Writing bin response:
>30   0x81 0x01 0x00 0x00
>30   0x00 0x00 0x00 0x00
>30   0x00 0x00 0x00 0x00
>30   0x87 0x90 0xa7 0xd9
>30   0x00 0x00 0x00 0x00
>30   0x00 0x00 0x00 0x01
30: going from conn_nread to conn_mwrite
30: going from conn_mwrite to conn_new_cmd
30: going from conn_new_cmd to conn_waiting
30: going from conn_waiting to conn_read
```

Figure 14.2 – Real world Memcached request and response headers

#1 Request (only headers are shown)
#2 Response

In figure 14.2, the highlighted section 2 is the response and section 1 represents a request to Memcached (only the request header is shown), which in this case is telling Memcached to SET the key "a" to the value "abc".

Each row in the highlighted sections represents 4 bytes; since there are 6 rows this means the request header is made up of 24 bytes as we said before. Looking back at table 14.1, you can begin to see how the header information from that table is represented in a real request. For now, this is all we need to know about the Memcached binary protocol. In the next section we need to take a look at just how we can start making these requests with Netty.

14.4 Netty Encoders and DEcoders

Netty is a complex and advanced framework but it's not psychic. When we make a request to set some key to a given value, we now know that an instance of the Request class is created to represent this request. This is great and works for us but Netty has no idea how this Request object translates to what Memcached expects. And the only thing Memcached expects is a series of bytes, regardless of the protocol you use, the data sent over the network is always a series of bytes.

To convert from a Request object to the series of bytes Memcached needs, Netty has what are known as encoders, so called because they encode things such as this `MemcachedRequest` class to another format. Notice I said another format, this is because encoders don't just encode from object to bytes, they can encode from one object to another type of object or from an object to a string and so on. Encoders are covered in greater detail later in chapter 7, including the various types Netty provides.

For now we're going to focus on the type that converts from an object to a series of bytes. To do this Netty offers an abstract class called `MessageToByteEncoder`. It provides an abstract method, which should convert a message (in this case our `MemcachedRequest` object) to bytes. You indicate what message your implementation can handle through use of Java's generics i.e. `MessageToByteEncoder<MemcachedRequest>` says this encoder encodes objects of the type `MemcachedRequest`.

MessageToByteEncoder and Generics

As said you use Generics to tell the MessageToByteEncoder to handle specific message type. If you want to handle many different message types with the same encoder you can also use MessageToByteEncoder<Object>. Just be sure to do proper instance checking of the message in this case.

All this is also true for decoders, except decoders convert a series of bytes back to an object .For this Netty provides the `ByteToMessageDecoder` class which instead of encode, provides a decode method. In the next two sections we'll see how we can implement a Memcached decoder and encoder. Before we do however, it's important to realize that you don't always need to provide your own encoders and decoders when using Netty. We're only doing it now because Netty does not have built in support for Memcached. If the protocol was HTTP or

another one of the many standard protocols Netty support then there would already be an encoder and decoder provided by Netty.

Encoder vs. Decoder

Remember encoder handles outbound and decoder inbound. Which basically means the encoder will encode data which is about to get written to the remote peer . The Decoder will handle data which was read from the remote peer.

It's important to remember there are two different directions related to outbound and inbound.

Be aware that our encoder and decoder do not verify any values for max size to keep the implementation simple. In a real world implementation you would most likely want to place in some verification checks and raise an `EncoderException` or `DecoderException` (or a sub-class of them) if there is protocol violation was detected.

14.4.1 Implementing the Memcached Encoder

This section puts our brief introduction to encoders into action. As said before the encoder is responsible to encode a message into a series of bytes. Those bytes can then be sent over the wire to the remote peer. To represent a request we ill first create the `MemcachedRequest` class, which we later then encode into a series of bytes with the actual encoder implementation.

Listing 14.1 shows our `MemcachedRequest` class, which represent the request.

Listing 14.1 – Implementation of a Memcached request

```
public class MemcachedRequest {                                    #1
    private static final Random rand = new Random();
    private int magic = 0x80;//fixed so hard coded
    private byte opCode; //the operation e.g. set or get
    private String key; //the key to delete, get or set
    private int flags = 0xdeadbeef; //random
    private int expires; //0 = item never expires
    private String body; //if opCode is set, the value
    private int id = rand.nextInt(); //Opaque
    private long cas; //data version check...not used
    private boolean hasExtras; //not all ops have extras

    public MemcachedRequest(byte opcode, String key, String value) {
        this.opCode = opcode;
        this.key = key;
        this.body = value == null ? "" : value;
        //only set command has extras in our example
        hasExtras = opcode == Opcode.SET;
    }

    public MemcachedRequest(byte opCode, String key) {
```

```
        this(opCode, key, null);
    }

    public int magic() {                                                #2
        return magic;
    }

    public int opCode() {                                               #3
        return opCode;
    }

    public String key() {                                               #4
        return key;
    }

    public int flags() {                                                #5
        return flags;
    }

    public int expires() {                                              #6
        return expires;
    }

    public String body() {                                              #7
        return body;
    }

    public int id() {                                                   #8
        return id;
    }

    public long cas() {                                                 #9
        return cas;
    }

    public boolean hasExtras() {                                       #10
        return hasExtras;
    }
}
```

#1 The class that represent a request which will be send to the Memcached server
#2 The magic number
ä3 The opCode which reflects for which operation the response was created
#4 The key for which the operation should be executed
#5 The extra flags used
#6 Indicate if the an expire should be used
#7 The body of any
#8 The id of the request. This id will be echoed back in the response.
#9 The compare-and-check value
#10 Returns true if any extras are used

The most important things about the `MemcachedRequest` class occur in the constructor, everything else is just there for support use later. The initialization that occurs here is reminiscent of what was shown in Table 14.1 earlier. In fact it is a direct implementation of the fields in table 14.1. These attributes are taken directly from the Memcached protocol specification.

For every request to set, get or delete, an instance of this `MemcachedRequest` class will be created. This means that should you want to implement the rest of the Memcached protocol you would only need to translate a "client.op*" (op* is any new operation you add) method to one of these requests. Before we move on to the Netty specific code, we need two more support classes.

Listing 14.2 – Possible Memcached operation codes and response statuses

```
public class Status {
    public static final short NO_ERROR = 0x0000;
    public static final short KEY_NOT_FOUND = 0x0001;
    public static final short KEY_EXISTS = 0x0002;
    public static final short VALUE_TOO_LARGE = 0x0003;
    public static final short INVALID_ARGUMENTS = 0x0004;
    public static final short ITEM_NOT_STORED = 0x0005;
    public static final short INC_DEC_NON_NUM_VAL = 0x0006;
}
public class Opcode {
    public static final byte GET = 0x00;
    public static final byte SET = 0x01;
    public static final byte DELETE = 0x04;
}
```

An `Opcode` tells Memcached which operations you wish to perform. Each operation is represented by a single byte. Similarly, when Memcached responds to a request, the response header contains two bytes, which represents the response status. The status and `Opcode` classes represent these Memcached constructs. Those `Opcodes` can be used when construct a new `MemcachedRequest` to specify which "Action" should be triggered by it.

But let us concentrate on the encoder for now., which will encode the previous created `MemcachedRequest` class in a series of bytes. For this we extend the `MessageToByteEncoder`, which is a perfect fit for this use-case.

Listing 14.3 shows the implementation in detail.

Listing 14.3 – MemcachedRequestEncoder implementation

```
public class MemcachedRequestEncoder extends
        MessageToByteEncoder<MemcachedRequest> {                          #1
    @Override
    protected void encode(ChannelHandlerContext ctx, MemcachedRequest msg,
                          ByteBuf out) throws Exception {
        // convert key and body to bytes array                            #2
        byte[] key = msg.key().getBytes(CharsetUtil.UTF_8);
        byte[] body = msg.body().getBytes(CharsetUtil.UTF_8);
        // total size of body = key size + content size + extras size     #3
        int bodySize = key.length + body.length + (msg.hasExtras() ? 8 : 0);

        // write magic byte                                               #4
        out.writeByte(msg.magic());
        // write opcode byte                                              #5
        out.writeByte(msg.opCode());
        // write key length (2 byte) i.e a Java short                     #6
```

```
            out.writeShort(key.length);
            // write extras length (1 byte)                                    #7
            int extraSize =  msg.hasExtras() ? 0x08 : 0x0;
            out.writeByte(extraSize);
            // byte is the data type, not currently implemented in
            // Memcached but required                                          #8
            out.writeByte(0);
            // next two bytes are reserved, not currently implemented
            // but are required                                                #9
            out.writeShort(0);

            // write total body length ( 4 bytes - 32 bit int)                #10
            out.writeInt(bodySize);
            // write opaque ( 4 bytes)  -  a 32 bit int that is returned
            // in the response                                                 #11
            out.writeInt(msg.id());

            // write CAS ( 8 bytes)
            // 24 byte header finishes with the CAS                            #12
            out.writeLong(msg.cas());

            if (msg.hasExtras()) {
                // write extras
                // (flags and expiry, 4 bytes each) - 8 bytes total           #13
                out.writeInt(msg.flags());
                out.writeInt(msg.expires());
            }
            //write key                                                       #14
            out.writeBytes(key);
            //write value                                                     #15
            out.writeBytes(body);
        }
    }
```

#1 The class that is responsible to encode the MemachedRequest into a series of bytes
#2 Convert the key and the actual body of the request into byte arrays.
#3 Calculate body size
#4 Write the magic as byte to the ByteBuf
#5 Write the opCode as byte
#6 Write the key length as short
#7 Write the extra length as byte
#8 Write the data type, which is always 0 as it is currently not used in Memcached but may be used in later versions
#9 Write in a short for reserved bytes which may be used in later versions of Memcached
#10 Write the body size as a long
#11 Write the opaque as int
#12 Write the cas as long. This is the last field of the header, after this the body starts
#13 Write the extra flags and expire as int if there are some present
#14 Write the key
#15 Write the body. After this the request is complete.

In summary, our encoder takes a request and using the `ByteBuf` Netty supplies, converts the `MemcachedRequest` into a correctly sequenced set of bytes.

In detail that is:

- Write magic byte

- Write opcode byte
- Write key length (2 byte)
- Write extras length (1 byte)
- Write data type (1 byte)
- Write null bytes for reserved bytes (2 bytes)
- Write total body length (4 bytes - 32 bit int)
- Write opaque (4 bytes) - a 32 bit int that is returned in the response
- Write CAS (8 bytes)
- Write extras (flags and expiry, 4 bytes each) - 8 bytes total
- Write key
- Write value

Whatever you put into the output buffer (the `ByteBuf` called out) Netty will send to the server the request is being written to. The next section will show how this works in reverse via decoders.

14.4.2 Implementing the Memcached Decoder

The same way we need to convert a `MemcachedRequest` object to a series of bytes, Memcached will only return bytes so we need to convert those bytes back into a response object that we can use in our application. This is where decoders come in.

So again we first create a POJO, which is used to represent the response in an easy to use manner.

Listing 14.7 - Implementation of a MemcachedResponse

```
public class MemcachedResponse {                                    #1

    private byte magic;
    private byte opCode;
    private byte dataType;
    private short status;
    private int id;
    private long cas;
    private int flags;
    private int expires;
    private String key;
    private String data;

    public MemcachedResponse(byte magic, byte opCode,
            byte dataType, short status, int id, long cas,
            int flags, int expires, String key, String data) {
        this.magic = magic;
        this.opCode = opCode;
        this.dataType = dataType;
        this.status = status;
        this.id = id;
```

```
        this.cas = cas;
        this.flags = flags;
        this.expires = expires;
        this.key = key;
        this.data = data;
    }

    public byte magic() {                                                #2
        return magic;
    }

    public byte opCode() {                                               #3
        return opCode;
    }

    public byte dataType() {                                             #4
        return dataType;
    }

    public short status() {                                              #5
        return status;
    }

    public int id() {                                                    #6
        return id;
    }

    public long cas() {                                                  #7
        return cas;
    }

    public int flags() {                                                 #8
        return flags;
    }

    public int expires() {                                               #9
       return expires;
    }

    public String key() {                                               #10
       return key;
    }

    public String data() {                                              #11
       return data;
    }
}
```

#1 The class that represent a response which was send back from the Memcached server
#2 The magic number
ä3 The opCode which reflects for which operation the response was created
#4 The data type which indicate if its binary or text based
#5 The status of the response which indicate of the request was successful etc
#6 The unique id
#7 The compare –and-set value
#8 The extra flags used
#9 Indicate if the value stored for this response will be expire at some point
#10 The key for which the response was created

#11 The actual data

The previous created `MemcachedResponse` class will now be used in our decoder to represent the response send back from the Memcached server. As we want to decode a series of bytes into a `MemcachedResponse` we will make use of the `ByteToMessageDecoder` base class.

Listing 14.4 shows the `MemcachedResponseDecoder` in detail.

Listing 14.4 – MemcachedResponseDecoder class

```
public class MemcachedResponseDecoder extends ByteToMessageDecoder {      #1

    private enum State {                                                   #2
        Header,
        Body
    }

    private State state = State.Header;
    private int totalBodySize;
    private byte magic;
    private byte opCode;
    private short keyLength;
    private byte extraLength;
    private byte dataType;
    private short status;
    private int id;
    private long cas;

    @Override
    protected void decode(ChannelHandlerContext ctx, ByteBuf in,
                          List<Object> out) {
        switch (state) {                                                   #3
            case Header:
                if (in.readableBytes() < 24) {
                    return;//response header is 24 bytes                   #4
                }
                // read header                                             #5
                magic = in.readByte();
                opCode = in.readByte();
                keyLength = in.readShort();
                extraLength = in.readByte();
                dataType = in.readByte();
                status = in.readShort();
                totalBodySize = in.readInt();
                id = in.readInt(); //referred to in the protocol spec as
opaque
                cas = in.readLong();

                state = State.Body;
                // fallthrough and start to read the body
            case Body:
                if (in.readableBytes() < totalBodySize) {
                    return; //until we have the entire payload return     #6
                }
                int flags = 0, expires = 0;
                int actualBodySize = totalBodySize;
```

```
                if (extraLength > 0) {                                      #7
                    flags = in.readInt();
                    actualBodySize -= 4;
                }
                if (extraLength > 4) {                                      #8
                    expires = in.readInt();
                    actualBodySize -= 4;
                }
                String key = "";
                if (keyLength > 0) {                                        #9
                    ByteBuf keyBytes = in.readBytes(keyLength);
                    key = keyBytes.toString(CharsetUtil.UTF_8);
                    actualBodySize -= keyLength;
                }
                ByteBuf body = in.readBytes(actualBodySize);               #10
                String data = body.toString(CharsetUtil.UTF_8);
                out.add(new MemcachedResponse(                             #11
                        magic,
                        opCode,
                        dataType,
                        status,
                        id,
                        cas,
                        flags,
                        expires,
                        key,
                        data
                ));

                state = State.Header;
            }

        }
    }
```

#1 The class that is responsible to create the MemcachedResponse out of the read bytes
#2 Represent current parsing state which means we either need to parse the header or body next
ä3 Switch based on the parsing state
#4 If not at least 24 bytes are readable it's impossible to read the whole header, so return here and wait to get notified again once more data is ready to be read
#5 Read all fields out of the header
#6 Check if enough data is readable to read the complete response body. The length was read out of the header before
#7 Check if there are any extra flags to read and if so do it
#8 Check if the response contains an expire field and if so read it
#9 check if the response contains a key and if so read it
#10 Read the actual body payload
#11 Construct a new MemachedResponse out of the previous read fields and data

So what happened in the implementation?

We know that a Memcached response has a 24 byte header, what we don't know is whether all the data that makes up a response will be included in the input `ByteBuf` when the decode method is called. This is because the underlying network stack may break the data into chunks. So to ensure we only decode when we have enough data, the above code checks if the amount of readable bytes available is at least 24 bytes. Once we have the first 24 bytes we can

determine how big the entire message is because this information is contained within the 24 bytes header.

When we've decoded an entire message, we create a `MemcachedResponse` and add it to the output list. Any object added to this list will be forwarded to the next `ChannelInboundHandler` in the `ChannelPipeline` and so allows to process it..

14.5 Testing the codec

With the previous created encoder and decoder we have our codec in place. But there is still something missing; Tests…

Without tests you will only see if it works when running it against some real server, which is not what you should depend on. As seen in chapter 10 writing tests for custom `ChannelHandler` is usually done via `EmbeddedChannel`.

So this is exactly what we do now to test our custom codec, which includes an encoder and decoder.

Let's us start with the encoder again. Listing 14.5 shows the simple written unit test.

Listing 14.5 – MemcachedRequestEncoderTest class

```
public class MemcachedRequestEncoderTest {

    @Test
    public void testMemcachedRequestEncoder() {
        MemcachedRequest request =
                new MemcachedRequest(Opcode.SET, "key1", "value1");      #1

        EmbeddedChannel channel = new EmbeddedChannel(
                new MemcachedRequestEncoder());                          #2
        Assert.assertTrue(channel.writeOutbound(request));               #3

        ByteBuf encoded = (ByteBuf) channel.readOutbound();

        Assert.assertNotNull(encoded);                                   #4
        Assert.assertEquals(request.magic(), encoded.readByte());        #5
        Assert.assertEquals(request.opCode(), encoded.readByte());       #6
        Assert.assertEquals(4, encoded.readShort());                     #7
        Assert.assertEquals((byte) 0x08, encoded.readByte());            #8
        Assert.assertEquals((byte) 0, encoded.readByte());               #9
        Assert.assertEquals(0, encoded.readShort());                    #10
        Assert.assertEquals(4 + 6 + 8, encoded.readInt());              #11
        Assert.assertEquals(request.id(), encoded.readInt());           #12
        Assert.assertEquals(request.cas(), encoded.readLong());         #13
        Assert.assertEquals(request.flags(), encoded.readInt());        #14
        Assert.assertEquals(request.expires(), encoded.readInt());      #15

        byte[] data = new byte[encoded.readableBytes()];                #16
        encoded.readBytes(data);
        Assert.assertArrayEquals((request.key() + request.body())
                .getBytes(CharsetUtil.UTF_8), data);
        Assert.assertFalse(encoded.isReadable());                       #17
```

```
        Assert.assertFalse(channel.finish());
        Assert.assertNull(channel.readInbound());
    }
}
```

#1 Create a new MemcachedRequest which should be encoded to a ByteBuf
#2 Create a new EmbeddedChannel which holds the MemcachedRequestEncoder to test
#3 Write the request to the channel and assert if it produced an encoded message
#4 Check if the ByteBuf is null
#5 Assert that the magic was correctly written into the ByteBuf
#6 Assert that the opCode (SET) was written correctly
#7 Check for the correct written length of the key
#8 Check if this request had extras included and so they was written
#9 Check if the data type was written
#10 Check if the reserved data was insert
#11 Check fo the total body size which is key.length + body.length + extras
#12 Check for correctly written id
#13 Check for correctly written Compare and Swap (CAS)
#14 Check for correct flags
#15 Check if expire was set
#16 Check if key and body are correct.

After tests for the encoder are in place, it's time to take care of the tests for the decoder. As we not know if the bytes will come in all in once or fragmented we need take special care to test both cases.

Listing 14.6 shows the tests in detail.

Listing 14.6 – MemcachedRequestEncoderTest class

```
public class MemcachedResponseDecoderTest {

    @Test
    public void testMemcachedResponseDecoder() {
        EmbeddedChannel channel = new EmbeddedChannel(
                new MemcachedResponseDecoder());                          #1

        byte magic = 1;
        byte opCode = Opcode.SET;
        byte dataType = 0;

        byte[] key = "Key1".getBytes(CharsetUtil.US_ASCII);
        byte[] body = "Value".getBytes(CharsetUtil.US_ASCII);
        int id = (int) System.currentTimeMillis();
        long cas = System.currentTimeMillis();

        ByteBuf buffer = Unpooled.buffer();                               #2
        buffer.writeByte(magic);
        buffer.writeByte(opCode);
        buffer.writeShort(key.length);
        buffer.writeByte(0);
        buffer.writeByte(dataType);
        buffer.writeShort(Status.KEY_EXISTS);
        buffer.writeInt(body.length + key.length);
        buffer.writeInt(id);
```

```
        buffer.writeLong(cas);
        buffer.writeBytes(key);
        buffer.writeBytes(body);

        Assert.assertTrue(channel.writeInbound(buffer));                #3

        MemcachedResponse response = (MemcachedResponse)
channel.readInbound();
        assertResponse(response, magic, opCode, dataType,
                Status.KEY_EXISTS, 0, 0, id, cas, key, body);           #4
    }

    @Test
    public void testMemcachedResponseDecoderFragments() {
        EmbeddedChannel channel = new EmbeddedChannel(
                new MemcachedResponseDecoder());                        #5

        byte magic = 1;
        byte opCode = Opcode.SET;
        byte dataType = 0;

        byte[] key = "Key1".getBytes(CharsetUtil.US_ASCII);
        byte[] body = "Value".getBytes(CharsetUtil.US_ASCII);
        int id = (int) System.currentTimeMillis();
        long cas = System.currentTimeMillis();

        ByteBuf buffer = Unpooled.buffer();                             #6
        buffer.writeByte(magic);
        buffer.writeByte(opCode);
        buffer.writeShort(key.length);
        buffer.writeByte(0);
        buffer.writeByte(dataType);
        buffer.writeShort(Status.KEY_EXISTS);
        buffer.writeInt(body.length + key.length);
        buffer.writeInt(id);
        buffer.writeLong(cas);
        buffer.writeBytes(key);
        buffer.writeBytes(body);

        ByteBuf fragment1 = buffer.readBytes(8);                        #7
        ByteBuf fragment2 = buffer.readBytes(24);
        ByteBuf fragment3 = buffer;

        Assert.assertFalse(channel.writeInbound(fragment1));            #8
        Assert.assertFalse(channel.writeInbound(fragment2));            #9
        Assert.assertTrue(channel.writeInbound(fragment3));            #10

        MemcachedResponse response = (MemcachedResponse)
channel.readInbound();
        assertResponse(response, magic, opCode, dataType,
                Status.KEY_EXISTS, 0, 0, id, cas, key, body);          #11
    }
```

```
    private static void assertResponse(MemcachedResponse response, byte
magic, byte opCode, byte dataType, short status, int expires, int flags, int
id, long cas, byte[] key, byte[] body) {
        Assert.assertEquals(magic, response.magic());
        Assert.assertArrayEquals(key,
response.key().getBytes(CharsetUtil.US_ASCII));
        Assert.assertEquals(opCode, response.opCode());
        Assert.assertEquals(dataType, response.dataType());
        Assert.assertEquals(status, response.status());
        Assert.assertEquals(cas, response.cas());
        Assert.assertEquals(expires, response.expires());
        Assert.assertEquals(flags, response.flags());
        Assert.assertArrayEquals(body,
response.data().getBytes(CharsetUtil.US_ASCII));
        Assert.assertEquals(id, response.id());
    }
}
```

#1 Create a new EmbeddedChannel which holds the MemcachedResponseDecoder to test
#2 Create a new Buffer and write data to it which match the structure of the binary protocol
#3 Write the buffer to the EmbeddedChannel and check if a new MemcachedResponse was created by assert the return value
#4 Assert the MemcachedResponse with the expected values
#5 Create a new EmbeddedChannel which holds the MemcachedResponseDecoder to test
#6 Create a new Buffer and write data to it which match the structure of the binary protocol
#7 Split the Buffer into 3 fragments
#8 Write the first fragment to the EmbeddedChannel and check that no new MemcachedResponse was created, as not all data is ready yet
#9 Write the second fragment to the EmbeddedChannel and check that no new MemcachedResponse was created, as not all data is ready yet
#10 Write the last fragment to the EmbeddedChannel and check that a new MemcachedResponse was created as we finally received all data.
#11 Assert the MemcachedResponse with the expected values

It's important to note that the shown tests don't provide full coverage but mainly "act" as example how you would make use of the `EmbeddedChannel` to test your custom written codec. How "complex" your tests need to be mainly depends on the implementation itself and so it's impossible to test what exactly you should test.

Anyway here are some things you should generally take care of:

- Testing with fragmented and non fragmented data
- Test validation if received data / send data if needed

14.6 Summary

After reading this chapter you should be able to create your own codec for your "favorite" protocol. This includes writing the encoder and decoder, which convert from bytes to your POJO and vise-versa. It shows how you can use a protocol specification and extract the needed information for the implementation.

Beside this it shows you how you complete your work by write unit tests for the encoder and decoder and so make sure everything works as expected without the need to run a full Memcached Server. This allows easy integration of tests into the build-system.

15

Choosing the right thread model

This chapter covers

- Details about the thread-model
- EventLoop
- Concurrency
- Task execution
- Task scheduling

The thread model defines how the application or framework executes your code, so it's important to choose the right thread model for the application/framework. Netty comes with an easy but powerful thread model that helps you simplify your code, because Netty takes care of all the needed synchronization in its core. All of your `ChannelHandlers`, which contain your business logic, are guaranteed to be executed by a single thread at the same time for a specific Channel. This doesn't mean Netty fails to use multithreading, but it does limit each connection to one thread as this design works well for nonblocking execution. This means there's no need for you to think about any of the issues you often have when working with multithreaded applications. You'll never again worry about `ConcurrentModificationException` and other problems such as stale data, locking, and so on, that you often have when developing with other frameworks.

After reading this chapter you'll have a deep understanding of Netty's thread model and why the Netty team chose it over other models. With this information you'll be able to get the best performance out of an application that uses Netty under the hood. Besides this, it will also help you keep your codebase clean, as the thread model makes it easy to write clean and simple code. You'll learn from the Netty team's experience, which used another thread model in the past and recently switched to the one explained here to make development with Netty even easier and more powerful.

Although this chapter covers Netty's internals, you can still use it for a general understanding of thread models and how to choose the perfect thread model for your next application.

This chapter assumes the following:

- You understand what threads are used for and have experience working with them.

  If that's not the case, please take time to gain this knowledge to make sure everything is clear to you. A good reference is *Java Concurrency in Practice* by Brian Goetz.

- You understand multithreaded applications and their designs.

  This also includes what it takes to keep them thread-safe while trying to get the best performance out of them.

- Knowledge of the `java.util.concurrent` package and its provided facilities via the `ExecutorService` and `ScheduledExecutorService` is a plus but not strictly needed.

15.1 Thread model overview

In this section you'll get a brief introduction to thread models in general, how Netty uses the specific thread model now, and how Netty used thread models in previous versions. You'll be better able to understand all the pros and cons of the different thread models.

If you think about it for a second, you may notice many situations in your normal life where you use a kind of thread model. To give you an analogy, think about it as a rule that enforces a behavior. Here's a real-world example.

You have a restaurant where you serve food to your customers. This food needs to be cooked in the kitchen before it's served. Once a customer orders food, you send a task to the kitchen that holds information that food X needs to be cooked. In your kitchen you can handle this in different ways—each of which would be like a thread model that defines how the task is executed.

- One cook prepares one meal at a time:

  This approach is single-threaded, which means only one task is executed at a time. After it's finished, the next order will get handled, and so on.

- You have multiple cooks. Each cook will cook the meal that needs to be prepared if he's not cooking anything else at the moment:

  This meal is multithreaded, as the tasks are handled by multiple threads (cooks). Many tasks can be executed at the same time.

- You have multiple cooks separated into groups. One group cooks dinner and one group cooks everything else:

  This situation is also multithreaded, but comes with extra restrictions. Many tasks are executed at the same time but are separated by the type of actual task (dinner vs. every other meal).

You see, even daily events fit well in a thread model. But how does Netty fit in here? Unfortunately it's not so simple, as Netty's core is multithreaded but hides most of this from the user. While Netty uses multiple threads to do all the work, only a single thread-like thread model is exposed to the user.

Before going into more detail, let's get a better understanding of the topic by revisiting what most applications do these days.

Most modern applications use more than one thread to dispatch work and therefore let the application use all of the system's available resources in an efficient manner. In the early days of Java, this was done by simply creating new threads on demand and starting them when work was required to be done in parallel.

But it soon turned out that this isn't perfect, because creating threads and recycling them comes with overhead. In Java 5 we finally got thread pools, which were defined by the Executor interface. Java 5 shipped many useful implementations, which vary significantly in their internals, but the idea is the same on all of them. They create threads and reuse them when a task is submitted for execution. This helps keep the overhead of creating and recycling threads to a minimum.

Figure 15.1 shows the how a thread pool is used to execute a task. It submits a task that will be executed by one free thread and once it's complete it frees up the thread again.

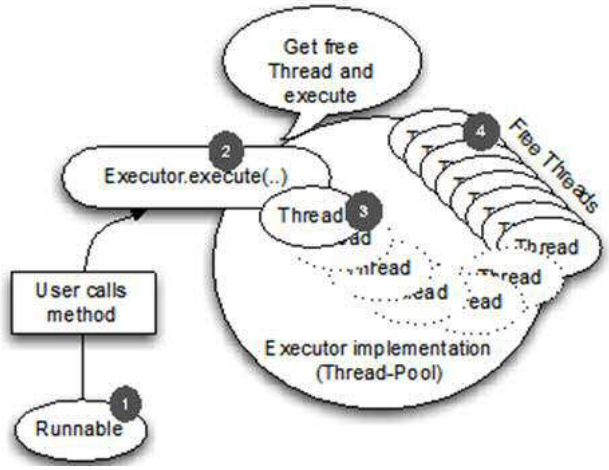


Figure 15.1 Executor execution logic

#1 Runnable which represents the task to execute. This could be anything from a database call to file system cleanup
#2 Previous runnable gets hand-over to the thread pool
#3 A free Thread is picked and the task is executed with it. When a thread has completed running it is put back into the list of free threads to be reused when a new task needs to be run
#4 Threads that execute tasks

This fixes the overhead of thread creation and recycling by not requiring new threads to be created and destroyed with each new task, but is only half of the problem, as we'll learn later.

You may ask yourself *why not use many threads all the time, now that an* `ExecutorService` *will help prevent the cost of creating and deallocating threads?*

Using multiple threads comes with the cost of managing resources and, as a side effect, introduces too much context switching. This gets worse with the number of threads that run and the number of tasks to execute. While using multiple threads may not seem like a problem at the beginning, it can hit you hard once you put real workload on the system.

In addition to these technical limitations and problems, other problems can occur that are more related to maintaining your application/framework in the future or during the lifetime of the project. It's valid to say that the complexity of your application rises depending on how parallel it is. To state it simply: Writing a multithreaded application is a hard job! What can we do to solve this problem? You need multiple threads to scale for real-world scenarios; that's a fact. Let's see how Netty solves this problem.

15.2 The event loop

The event loop does exactly what its name says. It runs events in a loop until it's terminated. This fits well in the design of network frameworks as they need to run events for a specific connection in a loop when these occur. This isn't something new that Netty invented; other frameworks and implementations have been doing this for ages.

The event loop is represented by the `EventLoop` interface in Netty. As an `EventLoop` extends from `EventExecutor`, which extends from `ScheduledExecutorService`, it's possible to also directly hand over a task that will get executed in the `EventLoop` on its next run.

EventExecutor , EventLoop and ScheduledExecutorService

Netty provides EventExecutor (which the class EventLoop extends) which extends Java's ScheduledExecutorService. So it is possible to hand a Task directly to an EventLoop instance and that task will be executed on the next run of the EventLoop.

The class/interface hierarchy of `EventExecutor` is shown in figure 15.2 (only shown in one depth).

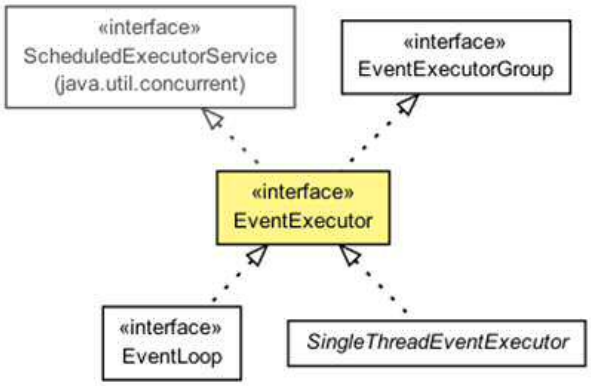


Figure 15.2 EventLoop class hierarchy

This interface makes it possible to use the provided `EventLoop` (which is assigned to a channel), as you'd typically use an `ExecutorService` to execute a task in one of the threads it holds.

15.2.1 Using the event loop

The following listing shows how to access the `EventLoop` that's assigned to a channel and use it to execute a task in the `EventLoop`.

Listing 15.1 Execute task in `EventLoop`

```
Channel ch = ...                                                    #1
Future<?> future = ch.eventLoop().execute(new Runnable() {         #2
    @Override
    public void run() {
System.out.println("Run in the EventLoop");                 #3
}
});
```

#1 Obtain reference to the channel
#2 Create runnable for the EventLoop accessed via channel and pass it to the EventLoop for execution. The task is executed as soon as the EventLoop is run again.
#3 Run the actual code here

The benefit of executing the task in the event loop is that you don't need to worry about any synchronization. The runnable will get executed in the same thread as all other events that are related to the channel. This fits exactly in the expected thread model of Netty.

To check if the task was executed yet, use the returned `Future`. This gives you access to many different operations.

The following listing verifies whether the task executed.

Listing 15.2 Execute task in event loop and check completion

```
Channel channel = ...
Future<?>future = channel.eventLoop().submit(…);#1
if (future.isDone()) {
    System.out.println("Task complete");                    #2
} else {
       System.out.println("Task not complete yet");            #3
}
```

#1 Obtain Future<?> returned by EventLoop
#2 At this point, task was executed
#3 At this point, task not executed, check later

To know whether a task will be executed directly, you may find it useful to check if you're in the `EventLoop`, as shown in the following listing. .

Listing 15.3 Execute task in `EventLoop`

```
Channel ch = ...
if (ch.eventLoop().inEventLoop()) {                              #1
System.out.println("In the EventLoop");                    #2
    ...
    ...
} else {
System.out.println("Outside the EventLoop");               #3
    ...
    ...
}
```

#1 Check if calling thread is assigned to EventLoop
#2 At this point, you're in the EventLoop
#3 At this point, thread isn't assigned to EventLoop

Shut down the thread pool only if you're sure that no other `EventLoop` is using it, or else you may get undefined side effects.

15.2.2 I/O operations in Netty 4

As this implementation is so powerful, even Netty uses it under the hood to handle its I/O events, which are triggered by read and write operations on the socket. These read and write operations are part of the network API that's provided by Java and the underlying operating system.

Figure 15.3 shows how inbound and outbound operations are executed in the context of the `EventLoop`. If the executing thread is bound to the `EventLoop` the operation will get executed directly. If not, the thread will be queued and executed once the `EventLoop` is ready.

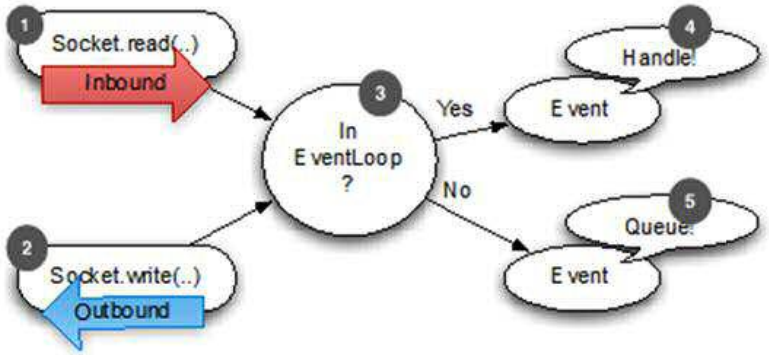


Figure 15.3 EventLoop handling of reads and writes

#1 Read from the socket, once data is received pass it to the EventLoop for further processing
#2 Handle writes to the socket. Once a write request is done, pass it to the EventLoop for further processing
#3 After the event is passed to the EventLoop, a check is performed to detect if the calling thread is the same as the one assigned to the EventLoop
#4 Thread is the same, so you're in the EventLoop, which means the event can get handled directly
#5 Thread is not the same as the one that's assigned to the EventLoop. Queue the event to get executed once the EventLoop process its events again

What exactly needs to be done once an event is handled depends on the nature of the event. Often it will read or transfer data from the network stack into your application. Other times it will do the same in the other direction, for example, transferring data from your application into the network stack (kernel) to send it over the remote peer. But it's not limited to this type of transaction; the important thing is that the logic used is generic and flexible enough to handle all kinds of use cases.

It should be noted that the thread model (on top of the event loop) described was not always used by Netty. In the next section you'll learn about the one in Netty 3. This will make it easier for you to understand why the new one is preferable.

15.2.3 I/O operations in Netty 3

In previous releases it was a bit different. Netty guaranteed only that inbound (previously called upstream) events were executed in the I/O thread (kind of an event loop). All outbound (previously called downstream) events were handled by calling the thread. This sounded like a good idea at first but turned out to be error-prone. It also put the burden on you to synchronize your `ChannelHandlers` that handle these events, as it wasn't guaranteed that only one thread operated on them at the same time. This could happen if you fired downstream events at the same time for one channel, for example, call `Channel.write(..)` in different threads. Figure 15.4 shows the execution flow of Netty 3.

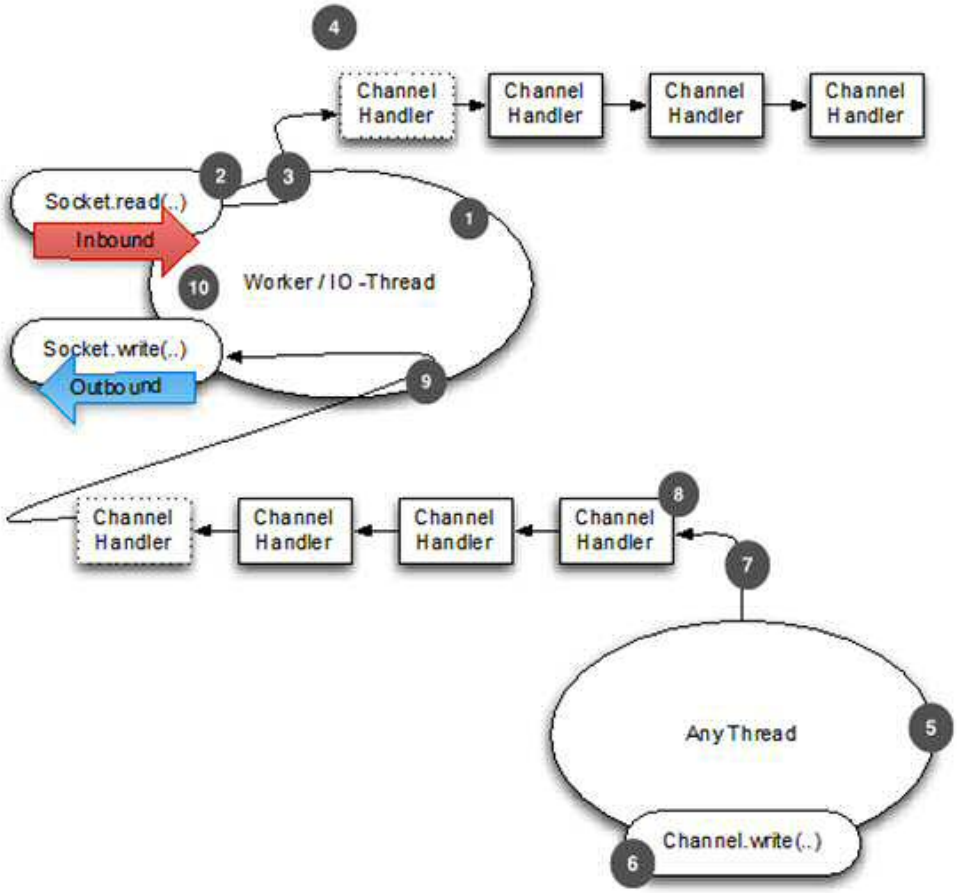


Figure 15.4 Execution-logic of Netty3

#1 The I/O thread that handles all the I/O events of a channel
#2 Data gets ready from the socket
#3 Gets processed in the worker, which is bound to the I/O thread
#4 The data / event is passed through all the ChannelHandlers of the Channel . This still happens within the IO thread of the Channel.
#5 Any thread from which an outbound operation is triggered. This may be the IO / thread or any other thread.
#6 Something is passed to the Channel.write(..)
#7 The write operation will generate an event that will get passed to the ChannelHandlers of the channel
#8 The event gets passed through all the ChannelHandlers of the channel This happens in the same thread as from which the write operation was triggerd.
#09 Once processing of the even via the ChannelHandlers is done, it will hand over the event to the worker thread
#10 The data is finally written to the remote peer by the worker thread

In addition to the burden placed on you to synchronize `ChannelHandlers`, another problematic side effect of this thread model is that you may need to fire an inbound (upstream) event as a result of an outbound (downstream) event. This is true, for example, if your `Channel.write(..)` operation causes an exception. In that case, an `exceptionCaught` must be generated and fired. This doesn't sound like a problem at first glance, but knowing that `exceptionCaught` is an inbound (upstream) event by design may give you an idea where the problem is. The problem is, in fact, that you now have a situation where the code is executed in your calling thread but the `exceptionCaught` event must be handed over to the worker thread for execution. That's generally doable, but if you forgot to hand it over, it will result in an invalidation of the thread model, which may give you all sorts of race conditions as the assumption about handing inbound (upstream) events only by one thread isn't true anymore.

The only positive effect of the previous implementation was that it could provide better latency in certain cases, but again, it's worth the cost as it removes complexity. In reality, in most applications you won't observe any difference in latency, which also depends on other factors such as:

- How fast the bytes can be written to the remote peer
- How busy the actual I/O thread is
- Context switching
- Locking

You can see that many details affecting the overall latency.

Now that you know how you can execute tasks in the event loop, it's time to have a quick look at various Netty internals that use this feature.

15.2.4 Netty's thread model internals

The trick that's used inside Netty to make its thread model perform so well is that it checks to see if the executing thread is the one that's assigned to the actual channel (and `EventLoop`). The `EventLoop` is responsible for handling all events for a `Channel` during its lifetime.

If the thread is the same as the one of the `EventLoop`, the code block in question is executed. If the thread is different, it schedules a task and puts it in an internal queue for later execution. It's usually executed once the next event for the `Channel` is handled via the `EventLoop`. This allows you to directly interact with the channel from any thread, while still being sure that all `ChannelHandlers` don't need to worry about concurrent access.

Figure 15.5 shows the execution logic that's used in the `EventLoop` to schedule tasks to fit Netty's thread model.

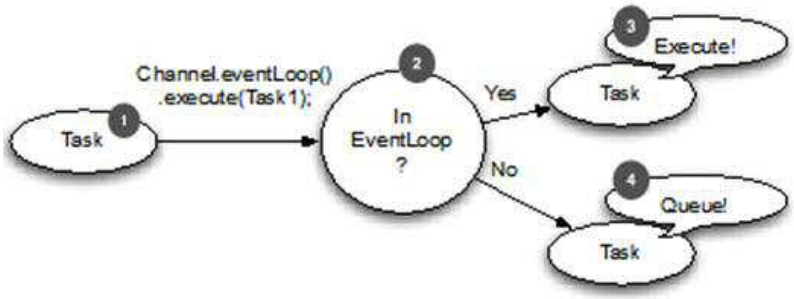


Figure 15.5 EventLoop execution logic / flow

#1 The task that should be executed in the EventLoop
#2 After the task is passed to the execute methods, a check is performed to detect if the calling thread is the same as the one that's assigned to the EventLoop
#3 The thread is the same, so you're in the EventLoop, which means the task can get executed directly
#4The thread is not the same as the one that's assigned to the EventLoop. Queue the task to get executed once the EventLoop processes its events again

That said, because of the design it's important to ensure that you never put any long-running tasks in the execution queue, because once the task is executed and run it will effectively block any other task from executing on the same thread. How much this affects the overall system depends on the implementation of the `EventLoop` that's used in the specific transport implementation.

As switching between transports is possible without any changes in your code base, it's important to remember that the Golden Rule always applies here: Never block the I/O thread. If you must do blocking calls (or execute tasks that can take long periods to complete), use a dedicated `EventExecutor`. More on this topic can be found in Chapter 6.

The next section will show one more feature that's often needed in applications. This is the need to schedule tasks for later execution or for periodic execution. Java has out-of-the-box solutions for this job, but Netty provides you with several advanced implementations that you'll learn about next.

15.3 Scheduling tasks for later execution

Every once in a while, you need to schedule a task for later execution. Maybe you want to register a task that gets fired after a client is connected for five minutes. A common use case is to send an "Are you alive?" message to the remote peer to see if it's still there. If it fails to respond, you know it's not connected anymore, and you can close the channel (connection) and release the resources.

Pretend you're on a phone call with your friend and after a period of silence you ask your friend if he's still on the call (to see if the line is broken). If he fails to respond to you, you know that either the call was interrupted, or he fell asleep. Whatever it is, you can quit the call,

too, because there's nothing for you to wait for. When you hung up, you released your resources and you're able to do something else.

The next section will show how you can schedule tasks for later execution in Netty using its powerful `EventLoop` implementation. It also gives you a quick introduction to task scheduling with the core Java API to make it easier for you to compare the built API with the one that comes with Netty. In addition to these items, you'll get more details about the internals of Netty's implementation and understand what advantages and what limitations it provides.

15.3.1 Scheduling tasks with plain Java API

To schedule a task you typically use the provided `ScheduledExecutorService` implementations that ship with the JDK. This wasn't always true. Before Java 5 you used a timer, which has the exact same limitations as normal threads.

Table 15.1 shows the utility methods that you can use to create an instance of `ScheduledExecutorService`.

Table 15.1 java.util.concurrent.Executors—Static methods to create a ScheduledExecutorService

Methods	Description
`newScheduledThreadPool(int corePoolSize)`	Creates a new `ScheduledThreadExecutorService` that can schedule commands to run after a delay or to execute periodically. It will use `corePoolSize` to calculate the number of threads.
`newScheduledThreadPool(int corePoolSize, ThreadFactorythreadFactory)`	
`newSingleThreadScheduledExecutor()`	Creates a new `ScheduledThreadExecutorService` that can schedule commands to run after a delay or to execute periodically. It will use one thread to execute the scheduled tasks.
`newSingleThreadScheduledExecutor(ThreadFactorythreadFactory)`	

After looking at the table 15.1 you'll notice that not many choices exist, but these are enough for most use cases. Now see how `ScheduledExecutorService` is used in listing 15.4 to schedule a task to run after a 60-second delay.

Listing 15.4 Schedule task with a `ScheduledExecutorService`

```
ScheduledExecutorService executor = Executors
.newScheduledThreadPool(10);                                    #1

ScheduledFuture<?> future = executor.schedule(
```

```
new Runnable() {                                          #2
    @Override
public void run() {
System.out.println("Now it is 60 seconds later"); #3
    }
}, 60, TimeUnit.SECONDS);                                 #4
...
...
executor.shutdown();                                      #5
```

#1 Create a new ScheduledExecutorService which uses 10 threads
#2 Create a new runnable to schedule for later execution
#3 This will run later
#4 Schedule task to run 60 seconds from now
#5 Shut down ScheduledExecutorService to release resources once task complete

As you can see, using `ScheduledExecutorService` is straightforward.

Now that you know how to use the classes of the JDK, you'll learn how you can use Netty's API to do the same, but in a more efficient way.

15.3.2 Scheduling tasks using EventLoop

Using the provided `ScheduledExecutorService` implementation may have worked well for you in the past, but it comes with limitations, such as the fact that tasks will be executed in an extra thread. This boils down to heavy resource usage if you schedule many tasks in an aggressive manner. This heavy resource usage isn't acceptable for a high-performance networking framework like Netty. What do you do if you must schedule tasks for later execution but still need to scale? Fortunately everything you need is already provided for free and part of the core API.

Netty solves this by allowing you to schedule tasks using the `EventLoop` that's assigned to the channel, as shown in listing 15.5.

Listing 15.5 Schedule task with `EventLoop`

```
Channel ch = ...
ScheduledFuture<?> future = ch.eventLoop().schedule(
new Runnable() {                  #1
    @Override
public void run() {
System.out.println("Now its 60 seconds later");       #2
}
}, 60, TimeUnit.SECONDS);                                 #3
```

#1 Create a new runnable to schedule for later execution
#2 This will run later
#3 Schedule task to run 60 seconds from now

After the 60 seconds passes, it will get executed by the `EventLoop` that's assigned to the channel.

As you learned earlier in the chapter, `EventLoop` extends `ScheduledExecutorService` and provides you with all the same methods that you learned to love in the past when using executors.

You can do other things like schedule a task that gets executed every X seconds. To schedule a task that will run every 60 seconds, use the code shown in listing 15.6.

Listing 15.6 Schedule a fixed task with the `EventLoop`

```
Channel ch = ...
ScheduledFuture<?> future = ch.eventLoop()
.scheduleAtFixedRate(new Runnable() {                    #1
    @Override
public void run() {
System.out.println("Run every 60 seconds");       #2
        }
}, 60, 60, TimeUnit.Seconds);#3
```

#1 Create new runnable to schedule for later execution
#2 This will run until the ScheduledFuture is canceled
#3 Schedule in 60 seconds and every 60 seconds

To cancel the execution, use the returned `ScheduledFuture` that's returned for every asynchronous operation. The `ScheduledFuture` provides a method for canceling a scheduled task or to check the state of it.

One simple cancel operation would look like the following example.

```
ScheduledFuture<?> future = ch.eventLoop()
.scheduleAtFixedRate(..);                              #1
// Some other code that runs...
future.cancel(false);                                        #2
```

#1 Schedule task and obtain the returned ScheduledFuture
#2 Cancel the task, which prevents it from running again

The complete list of all the operations can be found in the `ScheduledExecutorService` javadocs.

Now that you know what you can do with it, you may start to wonder how the implementation of the `ScheduledExecutorService` is different in Netty, and why it scales so well compared to other implementations of it.

15.3.3 Scheduling implementation internals

The actual implementation in Netty is based on the paper "Hashed and hierarchical timing wheels: Data structures to efficiently implement timer facility" by George Varghese. This kind of implementation only guarantees an approximated execution, which means that the execution of the task may not be 100% exactly on time. This has proven to be a tolerable limitation in practice and does not affect most applications at all. It's just something to remember if you need to schedule tasks, because it may not be the perfect fit if you need 100% exact, on-time execution.

To better understand how this works, think of it this way:

1. You schedule a task with a given delay.
2. The task gets inserted into the Schedule-Task-Queue of the `EventLoop`.
3. The `EventLoop` checks on every run if tasks need to get executed now.
4. If there's a task, it will execute it right away and remove it from the queue.
5. It waits again for the next run and starts over again with step 4.

Because of this implementation the scheduled execution may be not 100 % acurate. Which means it may not be 100 % exact, this is fine for most use-cases given that it allows for almost no overhead within Netty.

But what if you need more accurate execution? It's easy. You'll need to use another implementation of `ScheduledExecutorService` that's not part of Netty. Just remember that if you don't follow Netty's thread model protocol, you'll need to synchronize the concurrent access on your own. Do this only if you must.

15.4 I/O thread allocation in detail

Now that you know how to execute a task in the `EventLoop` or schedule a task for later execution, it's time to get into more detail about how Netty allocates threads.

Netty uses a thread pool to hold threads that will serve the I/O and events for a `Channel`. The way threads are allocated varies upon the transport implementation. An asynchronous implementation uses only a few threads that are shared between the channels. This allows a minimal number of threads to serve many channels (which maps 1:1 to a connection), eliminating the need to have one dedicated thread for each of them.

Figure 15.6 shows how the thread pool is allocated.

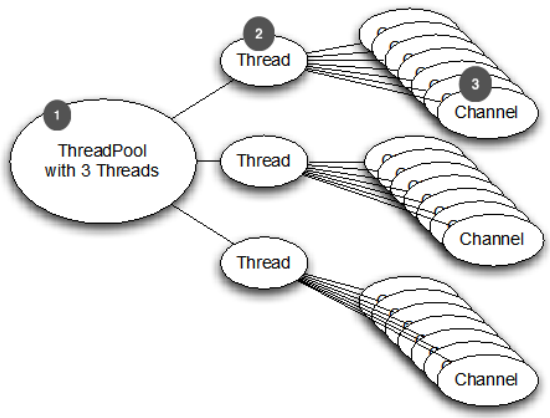


Figure 15.6 Thread allocation for nonblocking transports (such as NIO and AIO)

#1 All threads get allocated out of this thread pool. Here it will use three threads

#2 Thread allocated out of the thread pool. This thread handles all events and tasks for all the channels assigned to it, via the EventLoop
#3 Channels that are bound to the thread and so all operations are always executed by the same thread during the life-time of the Channel. A channel belongs to a connection

As you can see, figure 15.6 uses a thread pool with a fixed size of three threads. The threads are allocated directly once the thread pool is created. This is done to make sure resources are available when needed.

These three threads will get assigned to each newly created channel, which is created per connection. This is done through the `EventLoopGroup` implementation, which uses the thread pool under the hood to manage the resources. The actual implementation will take care of distributing the created channels evenly on all threads. This distribution is done in a round-robin fashion, so it may not be 100% accurate, but most of the time it is.

Once a `Channel` is assigned to a thread it will use this thread throughout its lifetime. This characteristic may change in future releases, so it may not be something that you should depend on. What won't change is the fact that only one thread will operate on the I/O of a specific channel at the same time. You can, and should, depend on this, as it makes sure that you don't need to worry at all about synchronization.

The semantic is a bit different for other transports, such as the shipped OIO (Old Blocking I/O) transport, as you can see in figure 15.7.

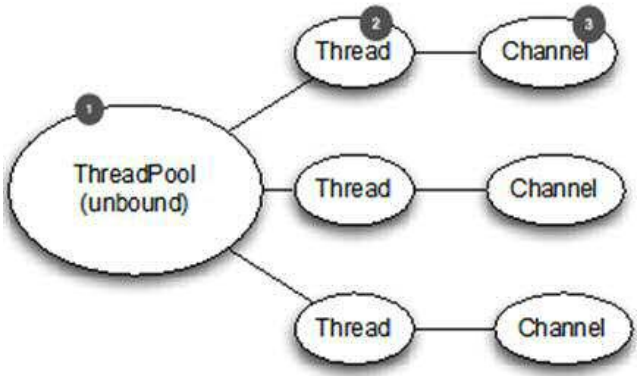


Figure 15.7 Thread allocation of blocking transports (such as OIO)

#1 All threads get allocated out of this thread pool
#2 Thread allocated for the channel will execute all events and task from EventLoop
#3 Channel bound to the thread. A channel belongs to a connection

As you may notice here, one thread is created per channel. You may be used to this from developing a network application that's based on regular blocking I/O when using classes out of the java.io.* package. But even if the semantics change in this case, one thing still stays the same. Each channel's I/O will only be handled by one thread at one time, which is the one

thread that powers the EventLoop of the Channel. You can depend on this hard rule; it's what makes writing code via the Netty framework so easy compared to other user network frameworks out there.

15.5 Summary

In this chapter you learned which thread model Netty uses. You learned the pros and cons of using thread models and how they simplify your life when working with Netty.

In addition to the inner workings, you also gained insight on how you can execute your own tasks in the event loop (I/O thread) the same way that Netty does. You learned how to schedule tasks for later execution and how this feature is implemented to scale even if you schedule a large number of tasks. You also learned how to verify whether a task has executed and how to cancel it.

You now know what thread model previous versions of Netty used, and you got more background information about why the new thread model is more powerful than the old one it replaced.

All of this should give you a deeper understanding of Netty's thread model, thereby helping you maximize your application's performance while minimizing the code needed. For more information about thread pools and concurrent access, please refer to the book *Java Concurrency in Practice* by Brian Goetz. His book will give you a deeper understanding of even the most complex applications that have to handle multithreaded use case scenarios.

16

De-register / Re-register with the EventLoop

In this chapter we'll take a look at

- `EventLoop`
- `Register and Deregistering from an EventLoop`
- `Using old Socket and Channels with Netty`

In the last chapter you learned about Netty's thread model and how this is implemented in the `EventLoop` of the specific Transports.

Netty provides an easy way to also attach `Socket` / `Channels` which were created outside of Netty and so "transfer" their responsibility to Netty. This allows you to integrate legacy frameworks in a seamless way and so make it easy to migrate to Netty step by step. Likewise Netty also allows you to deregister a `Channel` which will stop processing IO from it. This helps if you want to "suspend" and free up resources.

All of those are more advanced features, which you may never need, but never less it may become quite useful for you at some point. After this chapter you will be able to make use of those advanced features and so solve some harder problems.

As an example, imagine the case where you work with a popular social network and your application is growing amongst users. The current system handles a few thousand interactions/messages per second but if your growth continues the system will be processing tens of thousands of interactions per second.

This is exciting but the current system is inefficient, uses a lot of resources (memory and CPU), and management is convinced they can be saving some money whilst improving the service.

In cases like this, you cannot afford down time. The system must remain functional and processing the ever growing amount of data. This is a perfect use case for when registering/deregistering with an event loop becomes useful.

By allowing external sockets/channels to be registered and deregistered, Netty makes it possible for the old system to be plugged in such a way that all of Netty's benefits can become part of the existing system through an effective and delicate integration. The rest of this chapter will show you how integration can be achieved.

16.1 Registering and Deregistering Channels and Sockets

As explained in previous chapters each `Channel` needs to be registered on an `EventLoop` to process its IO / Events. This is done for you during the bootstrap process automatically.

Figure 16.1 shows the relationship.

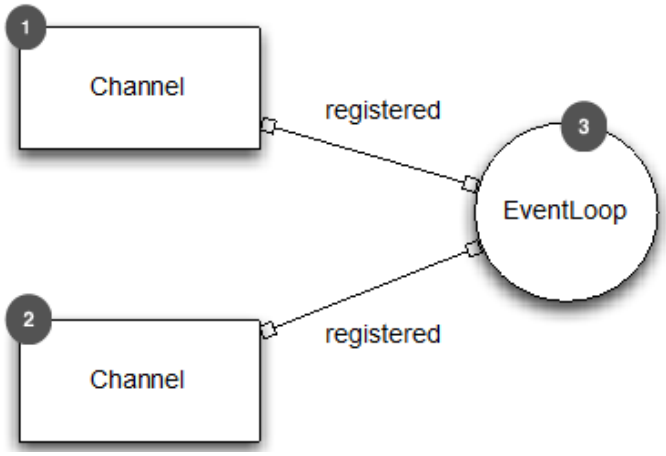


Figure 16.1 Bootstrap hierarchy

#1 A Channel which represent a connection with another peer
#2 A Channel which represent a connection with another peer
#3 The EventLoop which handles all the IO / Events for both Channels

But that's only one "half" of it. Once the `Channel` is closed it will also deregistered from its `EventLoop` again to free up resources.

As stated before sometimes you have to deal with `java.nio.channels.SocketChannel` or some other `java.nio.channels.Channel` implementation. This may be because you have some legacy app or framework that you need to use or any other reason.

The good news is that even with this you can make use of Netty as Netty allows to make use of pre-created `java.nio.channels.Channel` by wrapping them and then allows you to register on an `EventLoop`.

This allows you to make use of all Netty features while still re-using your existing stuff. Fortunately it's quite easy. Listing 16.1 shows how you can make use of this feature.

Listing 16.1 Register pre-created java.nio.channels.SocketChannel

```
java.nio.channels.SocketChannel mySocket =
java.nio.channels.SocketChannel.open();                          #1

// Perform some operations on the java.nio SocketChannel
...                                                              #2


SocketChannel ch = new NioSocketChannel(mySocket);               #3
EventLoopGroup group = ...;
ChannelFuture registerFuture = group.register(ch);               #4
...

// Deregister from EventLoop again.
ChannelFuture deregisterFuture = ch.deregister();                #5
```

#1 Create a java.nio.channels.SocketChannel via some legacy code
#2 Operate on the previous created java.nio.channels.SocketChannel
#3 Create a new NettySocketChannel by wrap the java.nio.channels.SocketChannel
#4 Register it with an EventLoop. After this Netty takes over and handle the IO events
#5 Deregister the SocketChannel from the EventLoop again. Netty does not handle any IO events for it anymore

This is all you need.

But the feature is not limited to `java.nio.channels.Channel` implementations. It also works for pre-created `Sockets`! The usage is quite the same; the only difference is that you would wrap the Socket into an `OioSocketChannel`. This is not surprising as the `OioSocketChannel` works in a blocking manner, which is the same as what `Socket` does.

To make it let stick have a look at Listing 16.2.

Listing 16.2 Register pre-created java.net.Socket

```
Socket mySocket = new Socket("www.manning.com", 80);             #1

// Perform some operations on the Socket
...                                                              #2


SocketChannel ch = new OioSocketChannel(mySocket);               #3
EventLoopGroup group = ...;
ChannelFuture registerFuture = group.register(ch);               #4
...

// Deregister from EventLoop again.
ChannelFuture deregisterFuture = ch.deregister();                #5
```

#1 Create a Socket , which is connected to the remote host via some legacy code
#2 Operate on the previous created Socket
#3 Create a new NettySocketChannel by wrap the java.nio.channels.SocketChannel
#4 Register it with an EventLoop. After this Netty takes over and handle the IO events
#5 Deregister the SocketChannel from the EventLoop again. Netty does not handle any IO events for it anymore

There are only two things to remember which are quite important.

1. If you use a pre-created Channel / Socket you will need to take care that no one else other than Netty operates on it once you register it. Otherwise you may see races or other problems.
2. The `EventLoop.register(…)` and `Channel.deregister(…)` operations are non-blocking a.k.a asynchronous, which means they may complete later. Because of this they return a `ChannelFuture`. So if you need to be sure the operation is done before do any further action make sure you either add a `ChannelFutureListener` or sync/ wait on the `ChannelFuture` to complete. Which one you use depends if you want / can block the calling thread or not but in general a channel future listener is recommend and blocking should be avoid unless absolutely necessary.

In this section you learned how you can make use of Netty's flexible `EventLoop` and register / deregister pre-created Channels on the fly. But there are also situations where you want to make use of this feature with Netty's own `Channels`.

Having the ability to deregister a `Channel` from its `EventLoop` provides you with some powerful and flexible way of dealing with `Channels` and their events. But what could it be helpful with or in fact, what is a valid use case for this?

There are a few of them but we will just focus on the two most needed to keep it simple and short.

16.2 Suspend IO Processing

There may be situations where you want to stop processing any events for a given `Channel`. This may be for various reasons; For example, it may be more crucial to keep an application from running out of memory and crashing than it is to lose a few messages. In a situation like this you would stop processing events for a channel, perform some cleanup operation and once the system is stabilized again, re-register the channel to continue processing messages.

The best way for doing so is to just deregister the `Channel` from its `EventLoop`. This will effectively stop any of event processing for the `Channel`.

Once you feel it is time to start processing again you can just re-register the `Channel` again on the `EventLoop` and everything will just continue. This means it will again read data from the socket and write data to it if needed. Beside this you will also be able to submit new tasks to execute for its `EventLoop`.

Figure 16.2 shows the use-case.

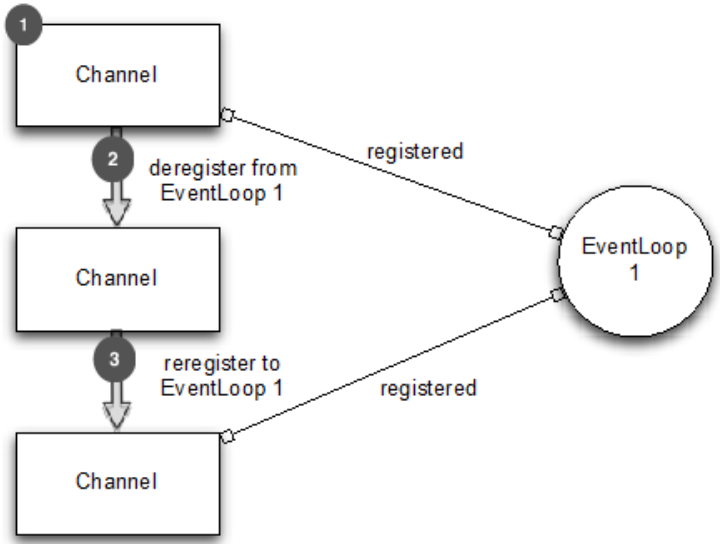


Figure 16.2 Deregister and reregister with an EventLoop

#1 A Channel which represent a connection with another peer
#2 Deregister the Channel from its EventLoop. After this is done no futher IO / Events are processed for the Channel
#3 Reregister the Channel with its previous EventLoop and so let the IO / Event processing start again.

Let us see it in action in Listing 16.3. In this example we will remove the `Channel` from its `EventLoop` as soon as the first data is received.

Listing 16.3 Register pre-created java.net.Socket

```
EventLoopGroup group = new NioEventLoopGroup();                          #1
Bootstrap bootstrap = new Bootstrap();
bootstrap.group(group).channel(NioSocketChannel.class)                   #2
    .handler(new SimpleChannelInboundHandler<ByteBuf>() {
         @Override
         protected void channelRead0(ChannelHandlerContext ctx,
                 ByteBuf byteBuf) throws Exception {
             ctx.pipeline().remove(this);
             ctx.deregister();                                           #3
         }
});
ChannelFuture future = bootstrap.connect(
new InetSocketAddress("www.manning.com", 80)).sync();

// Do something which takes some amount of time
...

// Register channel again on eventloop
```

```
Channel channel = future.channel();
group.register(channel).addListener(new ChannelFutureListener() {        #4
    @Override
    public void operationComplete(ChannelFuture future) throws Exception {
        if (future.isSuccess()) {
            System.out.println("Channel registered");
        } else {
            System.err.println("Register Channel on EventLoop failed");
            future.cause().printStackTrace();
        }
    }
});
```

#1 Createa new NioEventLoopGroup on which the Channel will be registered during the boostrap process
#2 Specify the EventLoopGroup to use
#3 Deregister the Channel from its EventLoop once data is received
#4 Register the Channel again on the EventLoop and so start to process IO / Events for it again

So what exactly happened in Listing 16.3? It's quite easy. First off we created the `Bootstrap` as usual and configured it with all needed parameters. If something is unclear here you may want to revisit Chapter 9, which gives a deep dive into bootstrapping. Now to the interesting part…

In our `SimpleChannelInboundHandler` which was given to the `Bootstrap.handler(…)` method we defined that it first remove itself from the `ChannelPipeline`(so it's a one "shot"). After the removal action was complete it calls `Channel.deregister()` which will deregister the `Channel` from its `EventLoop`. Once the Channel was removed from the `EventLoop` no more IO / Events are processed from it. This in fact means we've kind of suspended it. That is exactly what we wanted to do; suspend it to free up resources.

Now after some time has elapsed we want to "resume" the `Channel` and so it's IO/Event processing. For this the `Channel` is registered on the `EventLoopGroup` again. This is done via `EventLoopGroup.register(…)` in an asynchronous fashion. Once the operation completes the `Channel` is "in action" again.

16.3 Migrate a channel to another event loop

Another use case for deregister and register a `Channel` on the fly is to move an active `Channel` to another `EventLoop`.

There are many reasons why you may want to do this.

- The `EventLoop` may be to "busy" and you want to transfer the Channel to a "less-busy" one
- You want to terminate a `EventLoop` because you want to free up resources (in most cases that would be running Threads) while still keep the active `Channels` alive
- Migrate the `Channel` to a `EventLoop` which has "lower" execution priority based on some rules for "non-business-critical" data

Figure 16.3 shows the migration of the `Channel` to another `EventLoop`.

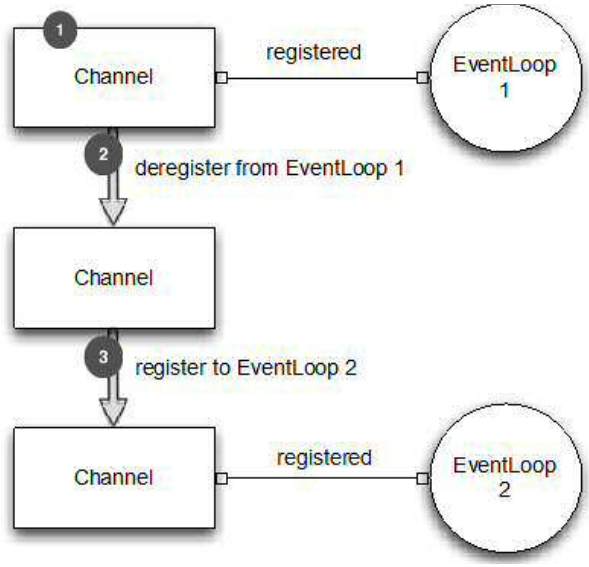


Figure 16.3 Migrate a Channel from one EventLoop to another

#1 A Channel which represent a connection with another peer
#2 Deregister the Channel from its EventLoop. After this is done no further IO / Events are processed for the Channel
#3 Register the Channel with another EventLoop and so let the IO / Event processing start again.

So let us see it again in action.

Listing 16.4 Register pre-created java.net.Socket

```
EventLoopGroup group = new NioEventLoopGroup();                          #1
final EventLoopGroup group2 = new NioEventLoopGroup();                   #2

Bootstrap bootstrap = new Bootstrap();
bootstrap.group(group).channel(NioSocketChannel.class)                   #3
    .handler(new SimpleChannelInboundHandler<ByteBuf>() {

        @Override
        protected void channelRead0(ChannelHandlerContext ctx,
                ByteBuf byteBuf) throws Exception {
            ctx.pipeline().remove(this);

            ChannelFuture cf = ctx.deregister();                         #4
            cf.addListener(new ChannelFutureListener() {

                @Override
                public void operationComplete(ChannelFuture future)
                        throws Exception {
                    group2.register(future.channel());                   #5
                }
```

```
            });
        }
    });

ChannelFuture future = bootstrap.connect(
new InetSocketAddress("www.manning.com", 80));
future.addListener(new ChannelFutureListener() {

    @Override
    public void operationComplete(ChannelFuture channelFuture)
            throws Exception {
        if (channelFuture.isSuccess()) {
            System.out.println("Connection established");
        } else {
            System.err.println("Connection attempt failed");
            channelFuture.cause().printStackTrace();
        }
    }
});
```

#1 Create a new NioEventLoopGroup on which the Channel will be registered during the bootstrap process
#2 Create another NioEventLoopGroup to which we want to migrate the Channel as soon as some data was received
#3Specify the EventLoopGroup to use
#4Deregister the Channel from its EventLoop once data is received
#5 Register the Channel to the other EventLoopGroup once the deregister completes and so start to process IO / Events for it again

As you saw migrating a `Channel` from one `EventLoop` to another is straightforward. The only thing you need to remember is that the `deregister(…)` and `register(…)` operations are asynchronous and so you need to operate with the `ChannelFuture` and `ChannelFutureListener` to make sure you do not try to register the Channel while it is still registered. If you try to do so you will trigger an `IllegalStateException` as a `Channel` can only be registered to one `EventLoop` at the same time.

16.4 Summary

This Chapter showed you how you can deregister and reregister a `Channel` with an `EventLoop`. This allows you to either "suspend" a `Channel` or migrate it to another `EventLoop` for any reason. Having the ability to register and deregister both pre-created and Netty channels makes many ad-hoc features a reality. It can be very useful for stabilizing a system and keeping it online by ignoring data/messages until some cleanup operation has been performed. Even more so, it can be very convenient for allowing you to progressively integrate legacy systems to start taking advantage of advanced Netty features with minimal risks to the old system.

In the next Chapter we will go much deeper in the Netty API and show you how you can implement your custom transport on top of Netty. This will allow you to reuse all the provided `ChannelHandler` that come with Netty but operate on your own transport implementation.

This will be the most advanced chapter and will teach you everything needed to become a real Netty expert.