The Netty Project 3.2 User Guide

The Proven Approach to Rapid Network Application Development

3.2.6.Final

Preface

1. The Problem

Nowadays we use general purpose applications or libraries to communicate with each other. For example, we often use an HTTP client library to retrieve information from a web server and to invoke a remote procedure call via web services.

However, a general purpose protocol or its implementation sometimes does not scale very well. It is like we don't use a general purpose HTTP server to exchange huge files, e-mail messages, and near-realtime messages such as financial information and multiplayer game data. What's required is a highly optimized protocol implementation which is dedicated to a special purpose. For example, you might want to implement an HTTP server which is optimized for AJAX-based chat application, media streaming, or large file transfer. You could even want to design and implement a whole new protocol which is precisely tailored to your need.

Another inevitable case is when you have to deal with a legacy proprietary protocol to ensure the interoperability with an old system. What matters in this case is how quickly we can implement that protocol while not sacrificing the stability and performance of the resulting application.

2. The Solution

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

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

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

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

Getting Started

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

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

1.1. Before Getting Started

The minimum requirements to run the examples which are introduced in this chapter are only two; the latest version of Netty and JDK 1.5 or above. The latest version of Netty is available in the project download page. To download the right version of JDK, please refer to your preferred JDK vendor's web site.

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

1.2. Writing a Discard Server

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

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

```
package org.jboss.netty.example.discard;

public class DiscardServerHandler extends SimpleChannelHandler {(1)

    @Override
    public void messageReceived(ChannelHandlerContext ctx, MessageEvent e) {(2)
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, ExceptionEvent e) {(3)
        e.getCause().printStackTrace();

        Channel ch = e.getChannel();
        ch.close();
    }
}
```

1. `DiscardServerHandler` extends `SimpleChannelHandler`, which is an implementation of `ChannelHandler`. `SimpleChannelHandler` provides various event handler methods that you can override. For now, it is just enough to extend `SimpleChannelHandler` rather than to implement the handler interfaces by yourself.

2. We override the `messageReceived` event handler method here. This method is called with a `MessageEvent`, which contains the received data, whenever new data is received from a client. In this example, we ignore the received data by doing nothing to implement the DISCARD protocol.

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

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

```java
package org.jboss.netty.example.discard;

import java.net.InetSocketAddress;
import java.util.concurrent.Executors;

public class DiscardServer {

    public static void main(String[] args) throws Exception {
        ChannelFactory factory =
            new NioServerSocketChannelFactory(1)(
                    Executors.newCachedThreadPool(),
                    Executors.newCachedThreadPool());

        ServerBootstrap bootstrap = new ServerBootstrap(2)(factory);

        bootstrap.setPipelineFactory(new ChannelPipelineFactory() {(3)
            public ChannelPipeline getPipeline() {
                return Channels.pipeline(new DiscardServerHandler());
            }
        });

        bootstrap.setOption("child.tcpNoDelay", true);(4)
        bootstrap.setOption("child.keepAlive", true);

        bootstrap.bind(new InetSocketAddress(8080));(5)
    }
}
```

1. ChannelFactory is a factory which creates and manages Channels and its related resources. It processes all I/O requests and performs I/O to generate ChannelEvents. Netty provides various ChannelFactory implementations. We are implementing a server-side application in this example, and therefore NioServerSocketChannelFactory was used. Another thing to note is that it does not create I/O threads by itself. It is supposed to acquire threads from the thread pool you specified in the constructor, and it gives you more control over how threads should be managed in the environment where your application runs, such as an application server with a security manager.

2. ServerBootstrap is a helper class that sets up a server. You can set up the server using a Channel directly. However, please note that this is a tedious process and you do not need to do that in most cases.

3. Here, we configure the ChannelPipelineFactory. Whenever a new connection is accepted by the server, a new ChannelPipeline will be created by the specified ChannelPipelineFactory. The new pipeline contains the DiscardServerHandler. As the application gets complicated, it is likely that you will add more handlers to the pipeline and extract this anonymous class into a top level class eventually.

4. You can also set the parameters which are specific to the Channel implementation. We are writing a TCP/IP server, so we are allowed to set the socket options such as tcpNoDelay and keepAlive. Please note that the "child." prefix was added to all options. It means the options will be applied to the accepted Channels instead of the options of the ServerSocketChannel. You could do the following to set the options of the ServerSocketChannel:

```
bootstrap.setOption("reuseAddress", true);
```

5. We are ready to go now. What's left is to bind to the port and to start the server. Here, we bind to the port 8080 of all NICs (network interface cards) in the machine. You can now call the bind method as many times as you want (with different bind addresses.)

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

1.3. Looking into the Received Data

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

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

We already know that MessageEvent is generated whenever data is received and the messageReceived handler method will be invoked. Let us put some code into the messageReceived method of the DiscardServerHandler:

```
@Override
public void messageReceived(ChannelHandlerContext ctx, MessageEvent e) {
    ChannelBuffer(1) buf = (ChannelBuffer) e.getMessage();
    while(buf.readable()) {
        System.out.println((char) buf.readByte());
        System.out.flush();
    }
}
```

(1) It is safe to assume the message type in socket transports is always `ChannelBuffer`. `ChannelBuffer` is a fundamental data structure which stores a sequence of bytes in Netty. It's similar to NIO `ByteBuffer`, but it is easier to use and more flexible. For example, Netty allows you to create a composite `ChannelBuffer` which combines multiple `ChannelBuffer`s reducing the number of unnecessary memory copy.

    Although it resembles to NIO `ByteBuffer` a lot, it is highly recommended to refer to the API reference. Learning how to use `ChannelBuffer` correctly is a critical step in using Netty without difficulty.

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

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

1.4. Writing an Echo Server

So far, we have been consuming data without responding at all. A server, however, is usually supposed to respond to a request. Let us learn how to write a response message to a client by implementing the ECHO protocol, where any received data is sent back.

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

```
@Override
public void messageReceived(ChannelHandlerContext ctx, MessageEvent e) {
    Channel(1) ch = e.getChannel();
    ch.write(e.getMessage());
}
```

(1) A `ChannelEvent` object has a reference to its associated `Channel`. Here, the returned `Channel` represents the connection which received the `MessageEvent`. We can get the `Channel` and call the `write` method to write something back to the remote peer.

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

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

1.5. Writing a Time Server

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

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

```
package org.jboss.netty.example.time;

public class TimeServerHandler extends SimpleChannelHandler {

    @Override
    public void channelConnected(ChannelHandlerContext ctx, ChannelStateEvent e) {1
        Channel ch = e.getChannel();

        ChannelBuffer time = ChannelBuffers.buffer(4);2
        time.writeInt((int) (System.currentTimeMillis() / 1000));

        ChannelFuture f = ch.write(time);3

        f.addListener(new ChannelFutureListener() {4
            public void operationComplete(ChannelFuture future) {
                Channel ch = future.getChannel();
                ch.close();
            }
        });
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, ExceptionEvent e) {
        e.getCause().printStackTrace();
        e.getChannel().close();
    }
}
```

1 As explained, `channelConnected` method will be invoked when a connection is established. Let us write the 32-bit integer that represents the current time in seconds here.

2 To send a new message, we need to allocate a new buffer which will contain the message. We are going to write a 32-bit integer, and therefore we need a `ChannelBuffer` whose capacity is 4 bytes. The `ChannelBuffers` helper class is used to allocate a new buffer. Besides the `buffer` method,

ChannelBuffers provides a lot of useful methods related to the ChannelBuffer. For more information, please refer to the API reference.

On the other hand, it is a good idea to use static imports for ChannelBuffers:

```
import static org.jboss.netty.buffer.ChannelBuffers.*;
...
ChannelBuffer  dynamicBuf = dynamicBuffer(256);
ChannelBuffer ordinaryBuf = buffer(1024);
```

3 As usual, we write the constructed message.

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

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

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

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

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

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

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

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

1.6. Writing a Time Client

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

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

```
package org.jboss.netty.example.time;

import java.net.InetSocketAddress;
import java.util.concurrent.Executors;

public class TimeClient {

    public static void main(String[] args) throws Exception {
        String host = args[0];
        int port = Integer.parseInt(args[1]);

        ChannelFactory factory =
            new NioClientSocketChannelFactory➀(
                    Executors.newCachedThreadPool(),
                    Executors.newCachedThreadPool());

        ClientBootstrap bootstrap = new ClientBootstrap➁(factory);

        bootstrap.setPipelineFactory(new ChannelPipelineFactory() {
            public ChannelPipeline getPipeline() {
                return Channels.pipeline(new TimeClientHandler());
            }
        });

        bootstrap.setOption("tcpNoDelay"➂, true);
        bootstrap.setOption("keepAlive", true);

        bootstrap.connect➃(new InetSocketAddress(host, port));
    }
}
```

➀ `NioClientSocketChannelFactory`, instead of `NioServerSocketChannelFactory` was used to create a client-side `Channel`.

➁ `ClientBootstrap` is a client-side counterpart of `ServerBootstrap`.

3 Please note that there's no `"child."` prefix. A client-side `SocketChannel` does not have a parent.

4 We should call the `connect` method instead of the `bind` method.

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

```
package org.jboss.netty.example.time;

import java.util.Date;

public class TimeClientHandler extends SimpleChannelHandler {

    @Override
    public void messageReceived(ChannelHandlerContext ctx, MessageEvent e) {
        ChannelBuffer buf = (ChannelBuffer) e.getMessage();
        long currentTimeMillis = buf.readInt() * 1000L;
        System.out.println(new Date(currentTimeMillis));
        e.getChannel().close();
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, ExceptionEvent e) {
        e.getCause().printStackTrace();
        e.getChannel().close();
    }
}
```

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

1.7. Dealing with a Stream-based Transport

1.7.1. One Small Caveat of Socket Buffer

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

```
+-----+-----+-----+
| ABC | DEF | GHI |
+-----+-----+-----+
```

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

```
+----+-------+---+---+
| AB | CDEFG | H | I |
+----+-------+---+---+
```

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

```
+-----+-----+-----+
| ABC | DEF | GHI |
+-----+-----+-----+
```

1.7.2. The First Solution

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

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

```
package org.jboss.netty.example.time;

import static org.jboss.netty.buffer.ChannelBuffers.*;

import java.util.Date;

public class TimeClientHandler extends SimpleChannelHandler {

    private final ChannelBuffer buf = dynamicBuffer();(1)

    @Override
    public void messageReceived(ChannelHandlerContext ctx, MessageEvent e) {
        ChannelBuffer m = (ChannelBuffer) e.getMessage();
        buf.writeBytes(m);(2)

        if (buf.readableBytes() >= 4) {(3)
            long currentTimeMillis = buf.readInt() * 1000L;
            System.out.println(new Date(currentTimeMillis));
```

```
                e.getChannel().close();
            }
        }

        @Override
        public void exceptionCaught(ChannelHandlerContext ctx, ExceptionEvent e) {
            e.getCause().printStackTrace();
            e.getChannel().close();
        }
    }
```

1. A *dynamic buffer* is a `ChannelBuffer` which increases its capacity on demand. It's very useful when you don't know the length of the message.

2. First, all received data should be cumulated into `buf`.

3. And then, the handler must check if `buf` has enough data, 4 bytes in this example, and proceed to the actual business logic. Otherwise, Netty will call the `messageReceived` method again when more data arrives, and eventually all 4 bytes will be cumulated.

1.7.3. The Second Solution

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

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

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

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

```
package org.jboss.netty.example.time;

public class TimeDecoder extends FrameDecoder(1) {

    @Override
    protected Object decode(
            ChannelHandlerContext ctx, Channel channel, ChannelBuffer buffer)(2) {

        if (buffer.readableBytes() < 4) {
            return null; (3)
        }
        return buffer.readBytes(4);(4)
    }
}
```

(1) FrameDecoder is an implementation of ChannelHandler which makes it easy to which deals with the fragmentation issue.

(2) FrameDecoder calls decode method with an internally maintained cumulative buffer whenever new data is received.

(3) If null is returned, it means there's not enough data yet. FrameDecoder will call again when there is a sufficient amount of data.

(4) If non-null is returned, it means the decode method has decoded a message successfully. FrameDecoder will discard the read part of its internal cumulative buffer. Please remember that you don't need to decode multiple messages. FrameDecoder will keep calling the decoder method until it returns null.

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

```
        bootstrap.setPipelineFactory(new ChannelPipelineFactory() {
            public ChannelPipeline getPipeline() {
                return Channels.pipeline(
                        new TimeDecoder(),
                        new TimeClientHandler());
            }
        });
```

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

```
package org.jboss.netty.example.time;

public class TimeDecoder extends ReplayingDecoder<VoidEnum> {

    @Override
    protected Object decode(
            ChannelHandlerContext ctx, Channel channel,
            ChannelBuffer buffer, VoidEnum state) {

        return buffer.readBytes(4);
    }
}
```

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

- `org.jboss.netty.example.factorial` for a binary protocol, and
- `org.jboss.netty.example.telnet` for a text line-based protocol.

1.8. Speaking in POJO instead of ChannelBuffer

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

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

First, let us define a new type called `UnixTime`.

```
package org.jboss.netty.example.time;

import java.util.Date;

public class UnixTime {
    private final int value;

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

    public int getValue() {
        return value;
    }

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

We can now revise the `TimeDecoder` to return a `UnixTime` instead of a `ChannelBuffer`.

```
@Override
protected Object decode(
        ChannelHandlerContext ctx, Channel channel, ChannelBuffer buffer) {
    if (buffer.readableBytes() < 4) {
        return null;
    }

    return new UnixTime(buffer.readInt());(1)
}
```

(1) `FrameDecoder` and `ReplayingDecoder` allow you to return an object of any type. If they were restricted to return only a `ChannelBuffer`, we would have to insert another `ChannelHandler` which transforms a `ChannelBuffer` into a `UnixTime`.

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

```
@Override
public void messageReceived(ChannelHandlerContext ctx, MessageEvent e) {
    UnixTime m = (UnixTime) e.getMessage();
    System.out.println(m);
    e.getChannel().close();
}
```

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

```
@Override
public void channelConnected(ChannelHandlerContext ctx, ChannelStateEvent e) {
    UnixTime time = new UnixTime(System.currentTimeMillis() / 1000);
    ChannelFuture f = e.getChannel().write(time);
    f.addListener(ChannelFutureListener.CLOSE);
}
```

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

```
package org.jboss.netty.example.time;

import static org.jboss.netty.buffer.ChannelBuffers.*;

public class TimeEncoder extends SimpleChannelHandler {

    public void writeRequested(ChannelHandlerContext ctx, MessageEvent(1) e) {
        UnixTime time = (UnixTime) e.getMessage();

        ChannelBuffer buf = buffer(4);
        buf.writeInt(time.getValue());

        Channels.write(ctx, e.getFuture(), buf);(2)
    }
}
```

(1) An encoder overrides the `writeRequested` method to intercept a write request. Please note that the `MessageEvent` parameter here is the same type which was specified in `messageReceived` but they are interpreted differently. A `ChannelEvent` can be either an *upstream* or *downstream* event depending on the direction where the event flows. For instance, a `MessageEvent` can be an upstream event when called for `messageReceived` or a downstream event when called for `writeRequested`. Please refer to the API reference to learn more about the difference between a upstream event and a downstream event.

(2) Once done with transforming a POJO into a `ChannelBuffer`, you should forward the new buffer to the previous `ChannelDownstreamHandler` in the `ChannelPipeline`. `Channels` provides various helper methods which generates and sends a `ChannelEvent`. In this example, `Channels.write(...)` method creates a new `MessageEvent` and sends it to the previous `ChannelDownstreamHandler` in the `ChannelPipeline`.

On the other hand, it is a good idea to use static imports for `Channels`:

```
import static org.jboss.netty.channel.Channels.*;
...
ChannelPipeline pipeline = pipeline();
write(ctx, e.getFuture(), buf);
fireChannelDisconnected(ctx);
```

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

1.9. Shutting Down Your Application

If you ran the `TimeClient`, you must have noticed that the application doesn't exit but just keep running doing nothing. Looking from the full stack trace, you will also find a couple I/O threads are running. To shut down the I/O threads and let the application exit gracefully, you need to release the resources allocated by `ChannelFactory`.

The shutdown process of a typical network application is composed of the following three steps:

1. Close all server sockets if there are any,
2. Close all non-server sockets (i.e. client sockets and accepted sockets) if there are any, and
3. Release all resources used by `ChannelFactory`.

To apply the three steps above to the `TimeClient`, `TimeClient.main()` could shut itself down gracefully by closing the only one client connection and releasing all resources used by `ChannelFactory`:

```
package org.jboss.netty.example.time;

public class TimeClient {
    public static void main(String[] args) throws Exception {
        ...
        ChannelFactory factory = ...;
        ClientBootstrap bootstrap = ...;
        ...
        ChannelFuture future(1) = bootstrap.connect(...);
        future.awaitUninterruptibly();(2)
        if (!future.isSuccess()) {
            future.getCause().printStackTrace();(3)
        }
        future.getChannel().getCloseFuture().awaitUninterruptibly();(4)
        factory.releaseExternalResources();(5)
    }
}
```

(1) The `connect` method of `ClientBootstrap` returns a `ChannelFuture` which notifies when a connection attempt succeeds or fails. It also has a reference to the `Channel` which is associated with the connection attempt.

(2) Wait for the returned `ChannelFuture` to determine if the connection attempt was successful or not.

(3) If failed, we print the cause of the failure to know why it failed. the `getCause()` method of `ChannelFuture` will return the cause of the failure if the connection attempt was neither successful nor cancelled.

(4) Now that the connection attempt is over, we need to wait until the connection is closed by waiting for the `closeFuture` of the `Channel`. Every `Channel` has its own `closeFuture` so that you are notified and can perform a certain action on closure.

Even if the connection attempt has failed the `closeFuture` will be notified because the `Channel` will be closed automatically when the connection attempt fails.

(5) All connections have been closed at this point. The only task left is to release the resources being used by `ChannelFactory`. It is as simple as calling its `releaseExternalResources()` method. All resources including the NIO `Selector`s and thread pools will be shut down and terminated automatically.

Shutting down a client was pretty easy, but how about shutting down a server? You need to unbind from the port and close all open accepted connections. To do this, you need a data structure that keeps track of the list of active connections, and it's not a trivial task. Fortunately, there is a solution, `ChannelGroup`.

`ChannelGroup` is a special extension of Java collections API which represents a set of open `Channel`s. If a `Channel` is added to a `ChannelGroup` and the added `Channel` is closed, the closed `Channel` is removed from its `ChannelGroup` automatically. You can also perform an operation on all `Channel`s in the same group. For instance, you can close all `Channel`s in a `ChannelGroup` when you shut down your server.

To keep track of open sockets, you need to modify the `TimeServerHandler` to add a new open `Channel` to the global `ChannelGroup`, `TimeServer.allChannels`:

```
@Override
public void channelOpen(ChannelHandlerContext ctx, ChannelStateEvent e) {
    TimeServer.allChannels.add(e.getChannel());①
}
```

① Yes, `ChannelGroup` is thread-safe.

Now that the list of all active `Channel`s are maintained automatically, shutting down a server is as easy as shutting down a client:

```
package org.jboss.netty.example.time;

public class TimeServer {

   static final ChannelGroup allChannels = new DefaultChannelGroup("time-server"①);

    public static void main(String[] args) throws Exception {
        ...
        ChannelFactory factory = ...;
        ServerBootstrap bootstrap = ...;
        ...
        Channel channel② = bootstrap.bind(...);
        allChannels.add(channel);③
        waitForShutdownCommand();④
        ChannelGroupFuture future = allChannels.close();⑤
        future.awaitUninterruptibly();
        factory.releaseExternalResources();
    }
}
```

① `DefaultChannelGroup` requires the name of the group as a constructor parameter. The group name is solely used to distinguish one group from others.

② The `bind` method of `ServerBootstrap` returns a server side `Channel` which is bound to the specified local address. Calling the `close()` method of the returned `Channel` will make the `Channel` unbind from the bound local address.

3 Any type of `Channel`s can be added to a `ChannelGroup` regardless if it is either server side, client-side, or accepted. Therefore, you can close the bound `Channel` along with the accepted `Channel`s in one shot when the server shuts down.

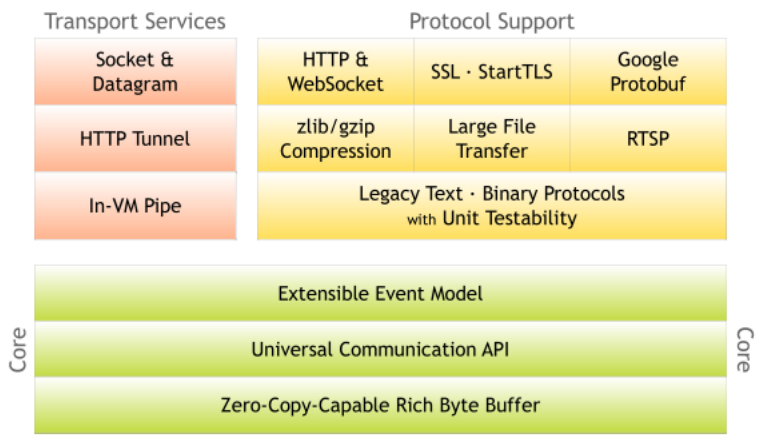
4 `waitForShutdownCommand()` is an imaginary method that waits for the shutdown signal. You could wait for a message from a privileged client or the JVM shutdown hook.

5 You can perform the same operation on all channels in the same `ChannelGroup`. In this case, we close all channels, which means the bound server-side `Channel` will be unbound and all accepted connections will be closed asynchronously. To notify when all connections were closed successfully, it returns a `ChannelGroupFuture` which has a similar role with `ChannelFuture`.

1.10. Summary

In this chapter, we had a quick tour of Netty with a demonstration on how to write a fully working network application on top of Netty. More questions you may have will be covered in the upcoming chapters and the revised version of this chapter. Please also note that the community is always waiting for your questions and ideas to help you and keep improving Netty based on your feed back.

Architectural Overview



In this chapter, we will examine what core functionalities are provided in Netty and how they constitute a complete network application development stack on top of the core. Please keep this diagram in mind as you read this chapter.

2.1. Rich Buffer Data Structure

Netty uses its own buffer API instead of NIO `ByteBuffer` to represent a sequence of bytes. This approach has significant advantages over using `ByteBuffer`. Netty's new buffer type, `ChannelBuffer` has been designed from the ground up to address the problems of `ByteBuffer` and to meet the daily needs of network application developers. To list a few cool features:

- You can define your own buffer type if necessary.
- Transparent zero copy is achieved by a built-in composite buffer type.
- A dynamic buffer type is provided out-of-the-box, whose capacity is expanded on demand, just like `StringBuffer`.
- There's no need to call `flip()` anymore.
- It is often faster than `ByteBuffer`.

For more information, please refer to the `org.jboss.netty.buffer` package description.

2.2. Universal Asynchronous I/O API

Traditional I/O APIs in Java provide different types and methods for different transport types. For example, `java.net.Socket` and `java.net.DatagramSocket` do not have any common super type and therefore they have very different ways to perform socket I/O.

This mismatch makes porting a network application from one transport to another tedious and difficult. The lack of portability between transports becomes a problem when you need to support additional transports, as this often entails rewriting the network layer of the application. Logically, many protocols can run on more than one transport such as TCP/IP, UDP/IP, SCTP, and serial port communication.

To make matters worse, Java's New I/O (NIO) API introduced incompatibilities with the old blocking I/O (OIO) API and will continue to do so in the next release, NIO.2 (AIO). Because all these APIs are different from each other in design and performance characteristics, you are often forced to determine which API your application will depend on before you even begin the implementation phase.

For instance, you might want to start with OIO because the number of clients you are going to serve will be very small and writing a socket server using OIO is much easier than using NIO. However, you are going to be in trouble when your business grows exponentially and your server needs to serve tens of thousands of clients simultaneously. You could start with NIO, but doing so may hinder rapid development by greatly increasing development time due to the complexity of the NIO Selector API.

Netty has a universal asynchronous I/O interface called a `Channel`, which abstracts away all operations required for point-to-point communication. That is, once you wrote your application on one Netty transport, your application can run on other Netty transports. Netty provides a number of essential transports via one universal API:

- NIO-based TCP/IP transport (See `org.jboss.netty.channel.socket.nio`),
- OIO-based TCP/IP transport (See `org.jboss.netty.channel.socket.oio`),
- OIO-based UDP/IP transport, and
- Local transport (See `org.jboss.netty.channel.local`).

Switching from one transport to another usually takes just a couple lines of changes such as choosing a different `ChannelFactory` implementation.

Also, you are even able to take advantage of new transports which aren't yet written (such as serial port communication transport), again by replacing just a couple lines of constructor calls. Moreover, you can write your own transport by extending the core API.

2.3. Event Model based on the Interceptor Chain Pattern

A well-defined and extensible event model is a must for an event-driven application. Netty has a well-defined event model focused on I/O. It also allows you to implement your own event type without breaking the existing code because each event type is distinguished from another by a strict type hierarchy. This is another differentiator against other frameworks. Many NIO frameworks have no or a very limited notion of an event model. If they offer extension at all, they often break the existing code when you try to add custom event types

A `ChannelEvent` is handled by a list of `ChannelHandler`s in a `ChannelPipeline`. The pipeline implements an advanced form of the Intercepting Filter pattern to give a user full control over how an event is handled and how the handlers in the pipeline interact with each other. For example, you can define what to do when data is read from a socket:

```
public class MyReadHandler implements SimpleChannelHandler {
    public void messageReceived(ChannelHandlerContext ctx, MessageEvent evt) {
        Object message = evt.getMessage();
        // Do something with the received message.
        ...

        // And forward the event to the next handler.
        ctx.sendUpstream(evt);
    }
}
```

You can also define what to do when a handler receives a write request:

```
public class MyWriteHandler implements SimpleChannelHandler {
    public void writeRequested(ChannelHandlerContext ctx, MessageEvent evt) {
        Object message = evt.getMessage();
        // Do something with the message to be written.
        ...

        // And forward the event to the next handler.
        ctx.sendDownstream(evt);
    }
}
```

For more information on the event model, please refer to the API documentation of `ChannelEvent` and `ChannelPipeline`.

2.4. Advanced Components for More Rapid Development

On top of the core components mentioned above, that already enable the implementation of all types of network applications, Netty provides a set of advanced features to accelerate the page of development even more.

2.4.1. Codec framework

As demonstrated in Section 1.8, “ Speaking in POJO instead of ChannelBuffer ”, it is always a good idea to separate a protocol codec from business logic. However, there are some complications when implementing this idea from scratch. You have to deal with the fragmentation of messages. Some protocols are multi-layered (i.e. built on top of other lower level protocols). Some are too complicated to be implemented in a single state machine.

Consequently, a good network application framework should provide an extensible, reusable, unit-testable, and multi-layered codec framework that generates maintainable user codecs.

Netty provides a number of basic and advanced codecs to address most issues you will encounter when you write a protocol codec regardless if it is simple or not, binary or text - simply whatever.

2.4.2. SSL / TLS Support

Unlike old blocking I/O, it is a non-trivial task to support SSL in NIO. You can't simply wrap a stream to encrypt or decrypt data but you have to use `javax.net.ssl.SSLEngine`. `SSLEngine` is a state machine which is as complex as SSL itself. You have to manage all possible states such as cipher suite and encryption key negotiation (or re-negotiation), certificate exchange, and validation. Moreover, `SSLEngine` is not even completely thread-safe, as one would expect.

In Netty, `SslHandler` takes care of all the gory details and pitfalls of `SSLEngine`. All you need to do is to configure the `SslHandler` and insert it into your `ChannelPipeline`. It also allows you to implement advanced features like StartTLS very easily.

2.4.3. HTTP Implementation

HTTP is definitely the most popular protocol in the Internet. There are already a number of HTTP implementations such as a Servlet container. Then why does Netty have HTTP on top of its core?

Netty's HTTP support is very different from the existing HTTP libraries. It gives you complete control over how HTTP messages are exchanged at a low level. Because it is basically the combination of an HTTP codec and HTTP message classes, there is no restriction such as an enforced thread model. That is, you can write your own HTTP client or server that works exactly the way you want. You have full control over everything that's in the HTTP specification, including the thread model, connection life cycle, and chunked encoding.

Thanks to its highly customizable nature, you can write a very efficient HTTP server such as:

- Chat server that requires persistent connections and server push technology (e.g. Comet and WebSockets)
- Media streaming server that needs to keep the connection open until the whole media is streamed (e.g. 2 hours of video)
- File server that allows the uploading of large files without memory pressure (e.g. uploading 1GB per request)
- Scalable mash-up client that connects to tens of thousands of 3rd party web services asynchronously

2.4.4. Google Protocol Buffer Integration

Google Protocol Buffers are an ideal solution for the rapid implementation of a highly efficient binary protocols that evolve over time. With `ProtobufEncoder` and `ProtobufDecoder`, you can turn the message classes generated by the Google Protocol Buffers Compiler (protoc) into Netty codec. Please take a look into the 'LocalTime' example that shows how easily you can create a high-performing binary protocol client and server from the sample protocol definition.

2.5. Summary

In this chapter, we reviewed the overall architecture of Netty from the feature standpoint. Netty has a simple, yet powerful architecture. It is composed of three components - buffer, channel, and event model - and all advanced features are built on top of the three core components. Once you understood how these three work together, it should not be difficult to understand the more advanced features which were covered briefly in this chapter.

You might still have unanswered questions about what the overall architecture looks like exactly and how each of the features work together. If so, it is a good idea to talk to us to improve this guide.