1

Communicating in Asynchronous World with Netty

In this chapter, we will cover asynchronous socket programming and common recipes using Netty for building network-based applications. The user story is, you get requirements to develop a system, that client and server can communicate asynchronously using TCP/IP . So we will divide the problem to 6 common recipes:

- Building an asynchronous TCP/IP server and client
- Sending hello message to server when connection is ready
- Receiving message asynchronously
- Get callback when closing an active Channel
- Get notification from ChannelHandler states
- Data pipeline processing with ChannelHandler

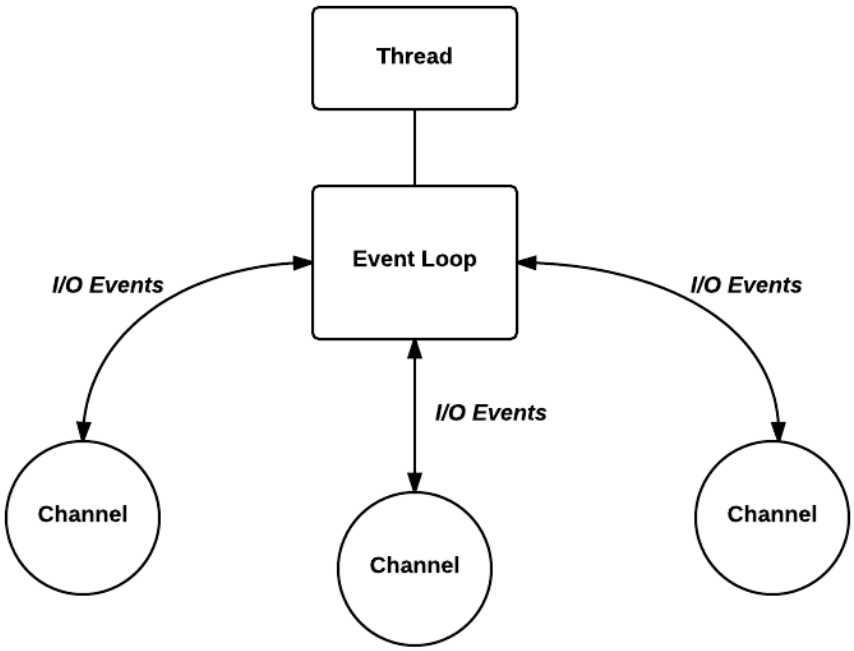
Introduction

Before we go to the first recipe, we need to ask ourselves the key question of the first chapter: Why is the asynchronous programming model more suitable for building applications in Internet-Scale Services ? And some main reasons are:

- Cooperatively schedule the CPU or other resources to each connection.
- Easily divide the work into relatively small pieces.
- Efficient and elegant coding style.
- Scalability - hundreds of connections means only hundreds of socket/state objects
- Requires no interlocking for access to shared resources.
- integrates easily with event-driven programming and Lambda Expression in Java 8

So we got some ideas to continue the next question “How Netty solves the asynchronous I/O?”

The answer is at the main loop, or the message dispatcher, is a programming construct that waits for and dispatches events or messages in a program. Your program at the Internet-Scale Services always receives a lot of data events, so with Netty, the work for building networking-based solutions is more elegant and easier.



Getting ready:

Make sure you install the Gradle Integration with Eclipse (4.4) and the JDK 8 is installed

Go to the page http://trieu.github.io/netty-cookbook

You will see the guideline on how to checkout the source code of the book.

Recipe 1.1 Building an asynchronous TCP Server

In this recipe, we go through a simple example to build a TCP server. You will use EventLoopGroup, ServerBootstrap, ChannelInitializer, StringEncoder, StringDecoder, ChannelHandler and ChannelInboundHandlerAdapter to build the solution.

How to do it ...

The basic main steps:

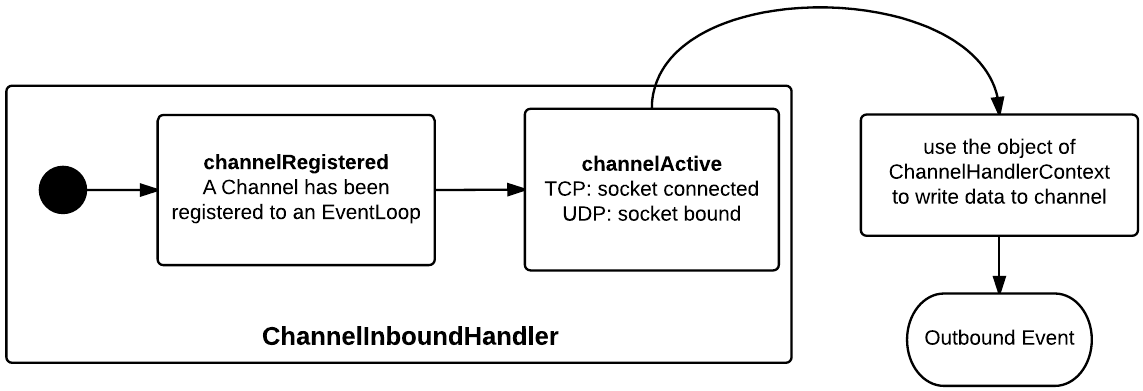
1. Configure the server with ServerBootstrap
2. Setting the String codec to define how server encoding/decoding messages and add to ChannelPipeline.
3. Implementing the handler, the subclass of ChannelInboundHandlerAdapter, that the main logic of our application
4. Start server and synchronize with the active channel, the socket that is listening

```
public final class AsynchTcpServer {
    static final int PORT = Integer.parseInt(System.getProperty("port", "8007"));
    public static void main(String[] args) throws Exception {
        EventLoopGroup bossGroup = new NioEventLoopGroup(1);
        EventLoopGroup workerGroup = new NioEventLoopGroup();
        try {
            ServerBootstrap b = new ServerBootstrap();
            b.group(bossGroup, workerGroup)
             .channel(NioServerSocketChannel.class)
             .handler(new LoggingHandler(LogLevel.INFO))
             .childHandler(new ChannelInitializer<SocketChannel>() {
                 @Override
                 public void initChannel(SocketChannel ch) throws Exception {
                     ChannelPipeline p = ch.pipeline();
                     p.addLast(new StringDecoder(CharsetUtil.UTF_8));
                     p.addLast(new StringEncoder(CharsetUtil.UTF_8));
                     p.addLast(new ChannelInboundHandlerAdapter(){
                       @Override
                      public void channelRead(ChannelHandlerContext ctx,
                                  Object msg) throws Exception {
                             String s = String.format("Response the message %s from
server", msg);
                             ctx.writeAndFlush(s);
                      }
```

```
                    });
                }
            });
            ChannelFuture f = b.bind(PORT).sync();
            Channel channel = f.channel();
            channel.closeFuture().sync();
        } finally {
            bossGroup.shutdownGracefully();
            workerGroup.shutdownGracefully();
        }
    }
}
```

Recipe 1.2 Sending hello message to server when connection is ready

In practice, sending messages between client and server is not safe. If the connection is not ready or not available, your code should be notified immediately if you want to send a message asynchronously to the server. You can extend the SimpleChannelInboundHandler in your implemented handler. Then, when a channel or a connection is ready, you can use the ChannelHandlerContext to send data.



How to do it …

The example code:

```
public class TcpClientHandler extends SimpleChannelInboundHandler<String> {
        String message;
        CallbackProcessor asynchCall;
    public TcpClientHandler(String message, CallbackProcessor asynchCall) {
        this.message = message;
        this.asynchCall = asynchCall;
    }
    @Override
    public void channelActive(ChannelHandlerContext ctx) throws Exception {
        ctx.writeAndFlush(this.message);
    }
```

Recipe 1.3 Receiving message asynchronously

You want to create a TCP client, and the response should be sent back to the caller. In the recipe, we can use the cool feature in Java 8, the Lambda Expression to create better elegant syntax likes this code

```
new TcpClient("127.0.0.1",8007).setMessage("Hello").onResponseReceived(rs -> {
        System.out.println(rs);
}).execute();
```

How to do it ...

Define a FunctionalInterface, named “CallbackProcessor”

```
@FunctionalInterface
public interface CallbackProcessor {
        public void process(String res);
}
```

In the handler, when the client receives the message from the server, the channelRead0 method is called and received data. In this code, you can apply the function “process” to delegate to the lambda method.

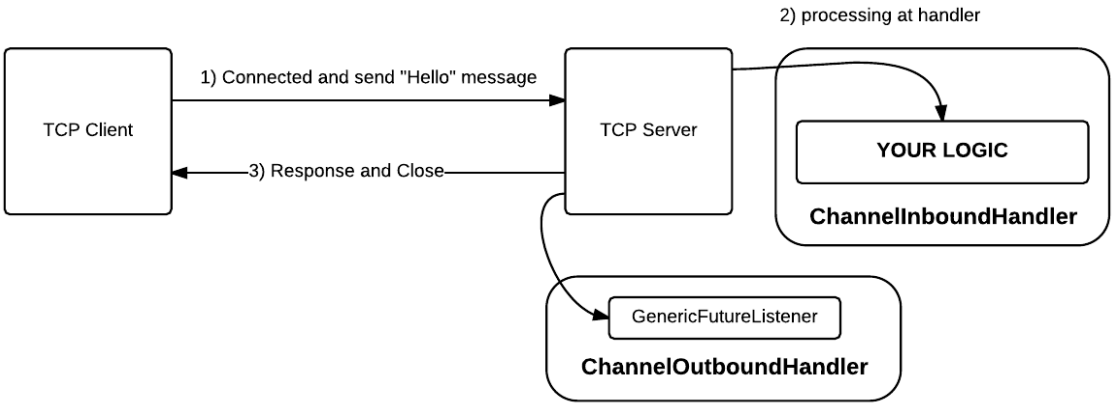
```
public class TcpClientHandler extends SimpleChannelInboundHandler<String> {
        String message;
        CallbackProcessor asynchCall;
    public TcpClientHandler(String message, CallbackProcessor asynchCall) {
        this.message = message;
        this.asynchCall = asynchCall;
    }
    @Override
    public void channelRead0(ChannelHandlerContext ctx, String msg) {
        this.asynchCall.process(msg);
    }
```

Recipe 1.4 Get notification from Netty I/O operations

Using the asynchronous I/O in Netty, or non-blocking I/O permits a ChannelHandler processing to continue before the transmission has finished. The advantages are improving throughput, latency, and/or responsiveness.

Channel I/O in Netty is designed to maximize CPU utilization and throughput by offloading most I/O onto a coprocessor, by using Zero-Copy Networking.

The context of our problem is when your application receives all data from peers, you want to close channel and add a listener to get notification when the operation completes.



How to do it …

Implement the interface GenericFutureListener to get notification when a TCP channel close successfully.

```
@Sharable
public class TcpServerOutboundHandler extends ChannelOutboundHandlerAdapter {
	@Override
	public void flush(ChannelHandlerContext ctx) throws Exception {
		super.flush(ctx);
		ctx.close().addListener(new GenericFutureListener<Future<? super
Void>>() {
			@Override
			public void operationComplete(Future<? super Void> future)
					throws Exception {
				System.out.println("close connection:
"+future.isSuccess());
			}
		});
	}
```

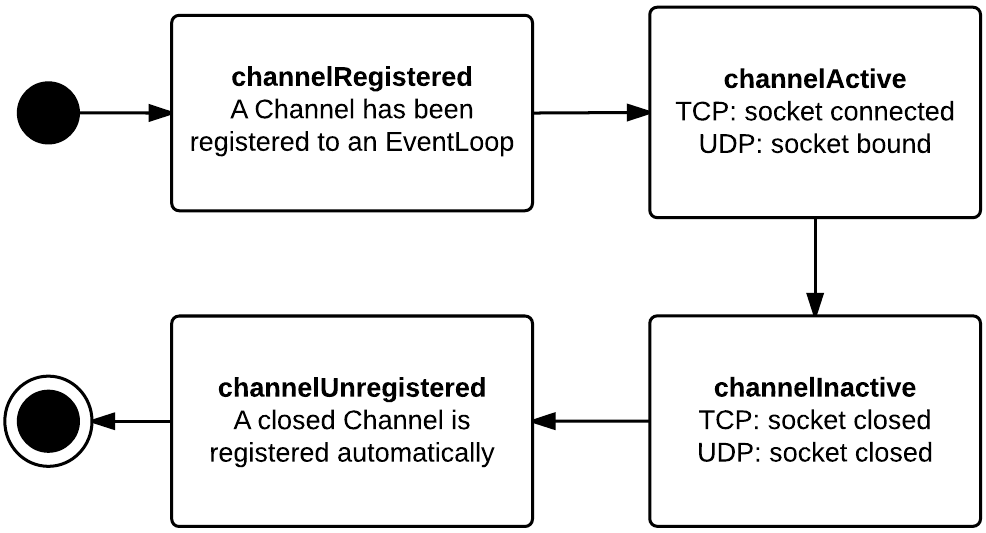
Recipe 1.5 Get notification from ChannelHandler states

In Netty, there are 2 core concepts that you usually use:

Channel : the link ('connection') to a resource that provides IO operations like read / write (the link between client and server)

ChannelHandler: Your business logic needs a place, and this is where it belongs too.

The state of the Channel is really important, because it tells you about the connection status. The diagram here , illustrate the state model of Channel



How can we hook the code and listen to the states of the Channel ?

How to do it …

In the interface ChannelInboundHandler, there are 4 methods you can override.

```
void channelRegistered(ChannelHandlerContext ctx) throws Exception;
void channelUnregistered(ChannelHandlerContext ctx) throws Exception;
void channelActive(ChannelHandlerContext ctx) throws Exception;
void channelInactive(ChannelHandlerContext ctx) throws Exception;
```

This example code for illustrating the solution:

```
public class TcpServerHandler extends ChannelInboundHandlerAdapter {
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) {
        System.out.println(msg);
         ctx.write("ok");
    }
    @Override
    public void channelRegistered(ChannelHandlerContext ctx) throws Exception {
        super.channelRegistered(ctx);
        InetSocketAddress address = (InetSocketAddress) ctx.channel().remoteAddress();
        System.out.println("channelRegistered "+ address.getAddress());
        isCatchedException = false;
    }
    @Override
    public void channelUnregistered(ChannelHandlerContext ctx) throws Exception {
        super.channelUnregistered(ctx);
        InetSocketAddress address = (InetSocketAddress) ctx.channel().remoteAddress();
        System.out.println("channelUnregistered "+ address.getAddress());
    }
    @Override
    public void channelActive(ChannelHandlerContext ctx) throws Exception {
        super.channelActive(ctx);
        System.out.println("channelActive "+ctx.channel());
        ctx.channel().writeAndFlush("connected");
    }
```

The output from code look likes this:

```
TcpServer (1) [Java Application] /usr/lib/jvm/jdk1.8.0_20/bin/java (Nov 15, 2014, 1:20:52 PM)
Nov 15, 2014 1:20:52 PM io.netty.handler.logging.LoggingHandler channelRegistered
INFO: [id: 0x5d022fac] REGISTERED
Nov 15, 2014 1:20:52 PM io.netty.handler.logging.LoggingHandler bind
INFO: [id: 0x5d022fac] BIND(0.0.0.0/0.0.0.0:8007)
Nov 15, 2014 1:20:52 PM io.netty.handler.logging.LoggingHandler channelActive
INFO: [id: 0x5d022fac, /0:0:0:0:0:0:0:0:8007] ACTIVE
Nov 15, 2014 1:20:59 PM io.netty.handler.logging.LoggingHandler logMessage
INFO: [id: 0x5d022fac, /0:0:0:0:0:0:0:0:8007] RECEIVED: [id: 0xe9ced320, /127.0.0.1:49462 => /127.0.0.1:8007]
channelRegistered /127.0.0.1
channelActive [id: 0xe9ced320, /127.0.0.1:49462 => /127.0.0.1:8007]
Hello
channelInactive /127.0.0.1:49462
channelUnregistered /127.0.0.1
```

Recipe 1.6 Data pipeline processing with ChannelHandler

In Netty, a ChannelPipeline is a set of data processing ChannelHandlers connected as pipe, where the output of one element is the input of the next one. The elements of a ChannelPipeline are often executed in parallel.

There are four concepts that you should familiar with in this recipe.

ChannelInboundHandler Defines listener methods for incoming I/O events

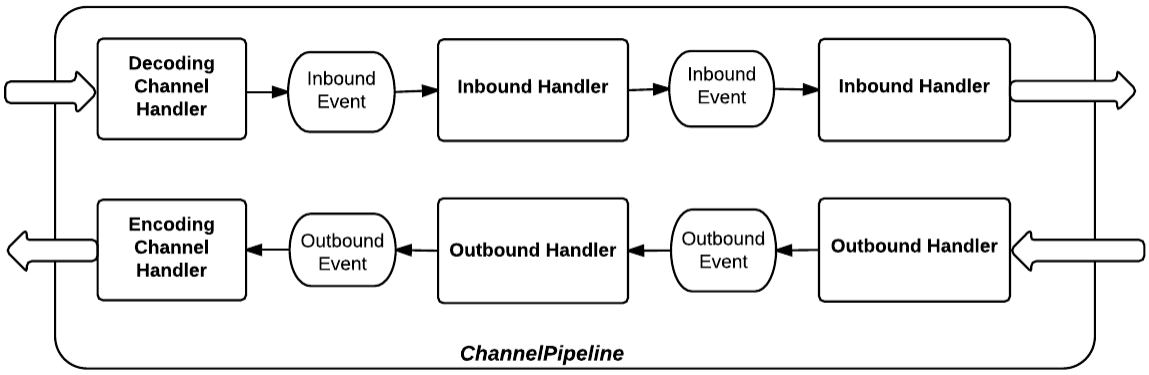
ChannelOutboundHandler Defines listener methods for outgoing I/O events

ChannelPipeline A list of ChannelHandlers that process incoming and outgoing events in a specific order.

ChannelInitializer : Configures the pipeline for a Channel.

The pattern in Netty normally likes this:

```
Channel ch = ...;
ChannelPipeline p = ch.pipeline();
EventExecutor e1 = new DefaultEventExecutor(16);
EventExecutor e2 = new DefaultEventExecutor(8);
p.addLast(new MyProtocolCodec());
p.addLast(e1, new MyDatabaseAccessingHandler());
p.addLast(e2, new MyHardDiskAccessingHandler());
```



The diagram above shows how inbound events and outbound events go through all handlers in ChannelPipeline.

How to do it …

```
public class DataProcessingPipeline extends ChannelInitializer<SocketChannel>{
        final static Logger logger = Logger.getLogger(DataProcessingPipeline.class);
        @Override
        protected void initChannel(SocketChannel ch) throws Exception {
                ChannelPipeline p = ch.pipeline();
          // the Decoder
          p.addLast("stringDecoder", new StringDecoder(CharsetUtil.UTF_8));
          // the Encoder
          p.addLast("stringEncoder", new StringEncoder(CharsetUtil.UTF_8));
          // the log handler and data transformer
          p.addLast("logger",new MessageToMessageDecoder<String>(){
                      @Override
                      protected void decode(ChannelHandlerContext ctx, String
msg,List<Object> out) throws Exception {
                              logger.info(String.format("logged raw data '%s'", msg));
                              InetSocketAddress address = (InetSocketAddress)
ctx.channel().remoteAddress();
                              Map<String, String> request = new HashMap<>();
```

```
                                request.put("data", msg);
                                request.put("from-ip",
address.getAddress().getHostAddress());
                                out.add(request);
                        }
                });
                // the processing logic handler
                p.addLast("handler",new SimpleChannelInboundHandler<Map<String, String>>(){
                      @Override
                     public void channelRead0(ChannelHandlerContext ctx, Map<String, String>
request)
                                throws Exception {
                        logger.info(String.format("from-host: '%s'",
request.get("from-ip")));
                        logger.info(String.format("data: '%s'", request.get("data")));
                        ctx.writeAndFlush("Done");
                     }
                });
        }
}
```

How it works…

The main idea is the implementation of MessageToMessageDecoder. The message “Hello” is sent from the client and is catched at StringDecoder first. Then, it transforms “ByteBuf” to String.

All classes , extends from abstract class MessageToMessageDecoder, the data will be catched at the method “protected void decode(ChannelHandlerContext ctx, String msg,List<Object> out)”. After processing the inbound event, data should be put into List<Object> out.

When you run the code, the output should look like this:

```
2014-11-17 14:41:45 INFO  LoggingHandler:100 - [id: 0x8daa6b57] REGISTERED
2014-11-17 14:41:45 INFO  LoggingHandler:100 - [id: 0x8daa6b57]
BIND(0.0.0.0/0.0.0.0:8007)
2014-11-17 14:41:45 INFO  LoggingHandler:100 - [id: 0x8daa6b57, /0:0:0:0:0:0:0:0:8007]
ACTIVE
2014-11-17 14:41:59 INFO  LoggingHandler:100 - [id: 0x8daa6b57, /0:0:0:0:0:0:0:0:8007]
RECEIVED: [id: 0x0fa09432, /127.0.0.1:47855 => /127.0.0.1:8007]
```

```
2014-11-17 14:41:59 INFO  DataProcessingPipeline:34 - logged raw data 'Hello'
2014-11-17 14:41:59 INFO  DataProcessingPipeline:47 - from-host: '127.0.0.1'
2014-11-17 14:41:59 INFO  DataProcessingPipeline:48 - data: 'Hello'
```