Netty 4 - Intro ⟶ Changes ⟶ HTTP ⟶ Lessons learned

@Apple iCloud 2014 · Cupertino · 2014/07/30

Norman Maurer, Principal Software Engineer / Leading Netty efforts
@ Red Hat Inc.


Netty / All things NIO

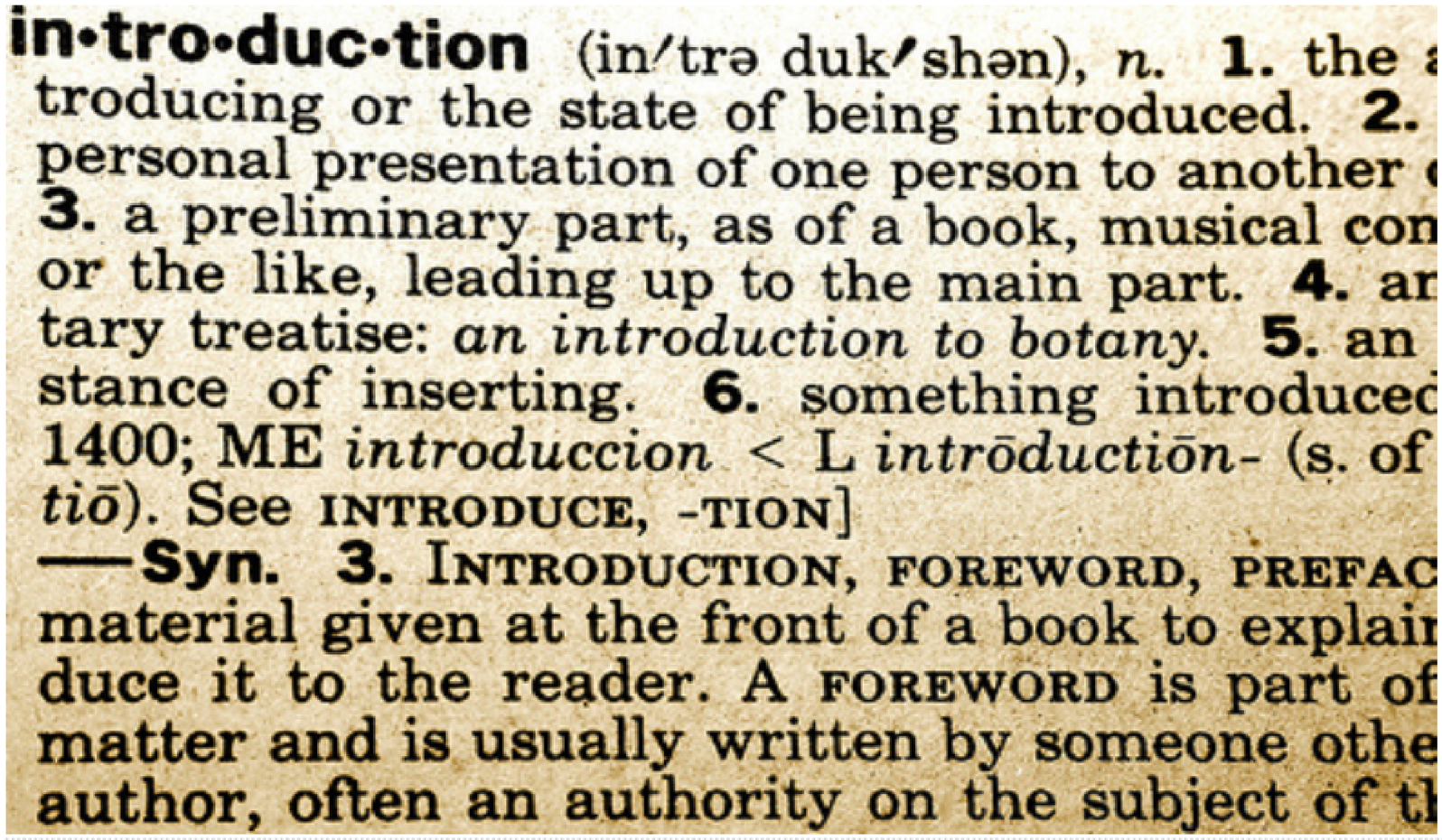

Author of Netty in Action


@normanmaurer


github.com/normanmaurer

- Introduction
- Design changes Netty 3 vs 4
- Netty for HTTP Server
- Lessons learned

Quick introduction


in•tro•duc•tion (in′trə duk′shən), *n.* **1.** the a
troducing or the state of being introduced. **2.**
personal presentation of one person to another
3. a preliminary part, as of a book, musical com
or the like, leading up to the main part. **4.** ar
tary treatise: *an introduction to botany.* **5.** an
stance of inserting. **6.** something introduced
1400; ME *introduccion* < L *intrōductiōn-* (s. of
tiō). See INTRODUCE, -TION]
—**Syn. 3.** INTRODUCTION, FOREWORD, PREFAC
material given at the front of a book to explai
duce it to the reader. A FOREWORD is part of
matter and is usually written by someone othe
author, often an authority on the subject of th

https://www.flickr.com/photos/larahsphotography/2795859728/

Fully asynchronous

- Asychronous from the ground up
- Using java.nio or native method calls for non-blocking io
- Futures and callbacks provided for easy composing

> *Don't call us, we'll call you.*
>
> – Hollywood principle

Hide complexity but not flexibility

- Hides all the complexity involved when you use `java.nio` or `java.nio2`
- Still empowers you with a lot of flexibility
- Unified API all over the place
- Allows easy testing of your custom code.

Protocol agnostic → Not just another HTTP server

- Supports `TCP`, `UDP`, `UDT`, `SCTP` out of the box
- Contains codecs for different protocols on top of these

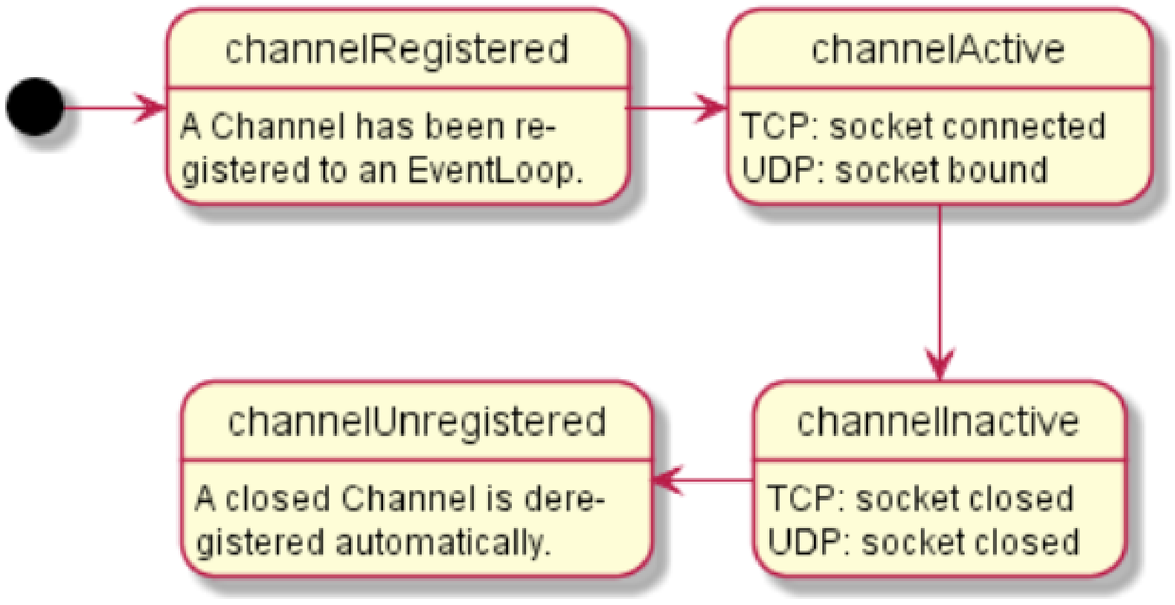
Thread-Model - Easy but powerful

- `Channel` is registered to `EventLoop` (1 x Thread) and all events are processed by the same Thread.
- One `EventLoop` will usually serve multiple `Channel` s
- Having inbound and outbound events handled by the same Thread simplifies concurrency handling a lot!

Simple state model

Allows to react on each state change by intercept the states via `ChannelHandler`.

Allows flexible handling depending on the needs.



ChannelPipeline

- Interceptor pattern
- Allows to add building-blocks (`ChannelHandler`) on-the-fly that transform data or react on events

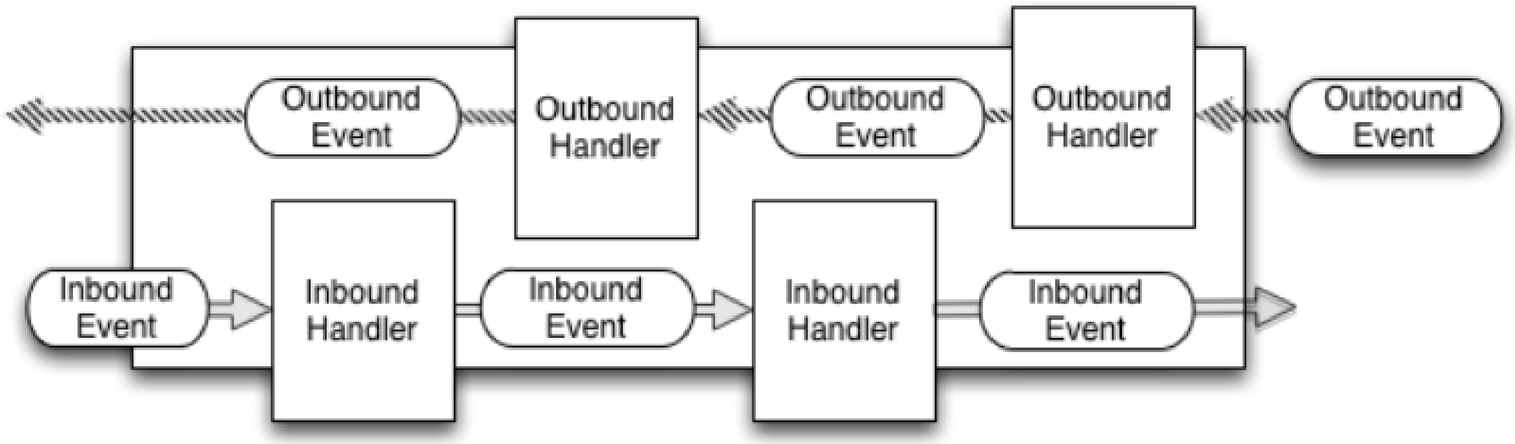
`Kind of a unix-pipe-like thing…`

```
$ echo "Netty is shit...." | sed -e 's/is /is the /' | cat (1)
Netty is the shit....
```

(1) Think of the whole line to be the `ChannelPipeline` and `echo`, `sed` and `cat` the `ChannelHandler` s that allow to transform data.

ChannelPipeline - How does it work

Inbound and outbound events flow through the `ChannelHandler` s in the `ChannelPipeline` and so allow to hook in.



ChannelPipeline - Compose processing logic

Compose complex processing logic via multiple `ChannelHandler`.

```
public class MyChannelInitializer extends ChannelInitializer<Channel> {
  @Override
  public void initChannel(Channel ch) {
    ChannelPipeline p = ch.pipeline();
    p.addLast(new SslHandler(...)); (1)
    p.addLast(new HttpServerCodec(...)); (2)
    p.addLast(new YourRequestHandler()); (3)
  }
}
```

1. Encrypt traffic
2. Support HTTP
3. Your handler that receive the HTTP requests.

ChannelHandler - React on received data

```
@Sharable
public class EchoHandler extends ChannelInboundHandlerAdapter {
  @Override
  public void channelRead(ChannelHandlerContext ctx, Object msg) { (1)
    ctx.writeAndFlush(msg);
  }

  @Override
  public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) { (2)
    cause.printStacktrace();
    ctx.close();
  }
}
```

(1) Intercept received message and write it back to the remote peer

(2) React on `Throwable` and close the connection

Decoder / Encoder - Transform data via ChannelHandler

Transform received ByteBuf to String

```
public class StringDecoder extends MessageToMessageDecoder<ByteBuf> {
  @Override
  protected void decode(ChannelHandlerContext ctx, ByteBuf msg, List<Object> out) {
      out.add(msg.toString(charset));
  }
}
```

Transform to be send String to ByteBuf

```
public class StringEncoder extends MessageToMessageEncoder<String> {
  @Override
  protected void encode(ChannelHandlerContext ctx, String msg, List<Object> out) {
    out.add(ByteBufUtil.encodeString(ctx.alloc(), CharBuffer.wrap(msg), charset));
  }
}
```

Adding other processing logic?

Adding more processing logic is often just a matter of adding just-another `ChannelHandler` to the `ChannelPipeline`.

Design Changes

https://www.flickr.com/photos/nhussein/3833334809

Thread-Model…

- Netty 3: Inbound ⇒ IO-Thread , Outbound ⇒ calling Thread :(
- Netty 4: Inbound / Outbound ⇒ IO-Thread
- Having inbound and outbound handled by the IO-Thread simplifies concurrency handling a lot!

Events vs. direct message invocation…

Netty 3: Create new `ChannelEvent` for each IO event.

Netty 4: Use dedicated method invocation per event.

http://25.media.tumblr.com/tumblr_me2eq0PnBx1rtu0cpo1_1280.jpg

Reduces GC-Pressure a lot!

ChannelHandler - Less confusing naming

Inbound:

- Netty 3: `ChannelUpstreamHandler`
- Netty 4: `ChannelInboundHandler`

Outbound:

- Netty 3: `ChannelDownstreamHandler`
- Netty 4: `ChannelOutboundHandler`

Pass custom events through `ChannelPipeline`

Your custom events

```
public enum CustomEvents {
  MyCustomEvent
}

public class CustomEventHandler extends ChannelInboundHandlerAdapter {
  @Override
  public void userEventTriggered(ChannelHandlerContext ctx, Object evt) {
    if (evt == MyCustomEvent) { // do something}
  }
}

ChannelPipeline pipeline = channel.pipeline();
pipeline.fireUserEventTriggered(MyCustomEvent);
```

Good fit for handshake notifications and more

Write behaviour

Netty 3: `Channel.write(...)` ⇒ write to socket via syscall

Netty 4: `Channel.write(...)` ⇒ write through the pipeline but DOESN'T trigger syscall. To trigger write to socket use `Channel.flush()`

Gives more flexibility for when things are written and also allows efficient pipelining.

Split ChannelFuture into ChannelFuture and ChannelPromise

Netty 3: `ChannelFuture` allowed to be notified directly via `setSuccess()` etc..

Netty 4: `ChannelFuture` only allows to receive notifications. `ChannelPromise` allows to change state.

Clearer who is responsible to notify a future and who is not.

EventLoopGroup to rule them all

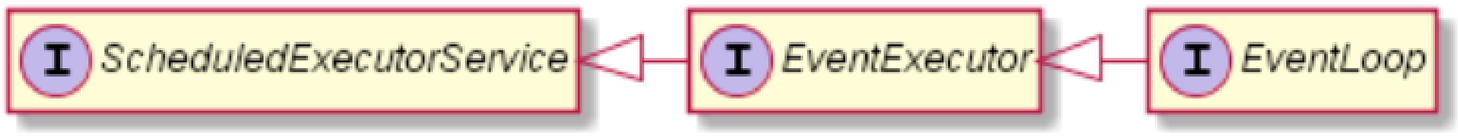
- `EventLoopGroup` used for boss and worker
- Can share `EventLoopGroup` between server and client to minimize threads and latency
- Register `EventLoop` to `Channel`

Share EventLoops

```
EventLoopGroup group = new NioEventLoopGroup();
Bootstrap cb = new Bootstrap();
cb.group(group);

ServerBootstrap sb = new ServerBootstrap();
sb.group(group);
```

EventLoop - All the `ScheduleExecutorService` goodies for free!




```
public class WriteTimeOutHandler extends ChannelOutboundHandlerAdapter {
  @Override
  public void write(ChannelHandlerContext ctx, Object msg, ChannelPromise promise) {
    ctx.write(msg, promise);

    if (!promise.isDone()) {
      ctx.executor().schedule(new WriteTimeoutTask(promise), 30, TimeUnit.SECONDS); (1)
    }
  }
}
```

(1) Schedule task for in 30 seconds

Execute ChannelHandler outside of EventLoop

- Netty 3: Use `OrderedMemoryAwareThreadPoolExecutor`
- Netty 4: Support build into the `ChannelPipeline`

```
Channel ch = ...;
ChannelPipeline p = ch.pipeline();
EventExecutor e1 = new DefaultEventExecutor(16);
p.addLast(new MyProtocolCodec()); (1)
p.addLast(e1, new MyDatabaseAccessingHandler()); (2)
```

(1) Executed in `EventLoop` (and so the `Thread` bound to it)

(2) Executed in one of the `EventExecutors` of `e1`

Buffers

Netty 3: Always use heap buffers by default

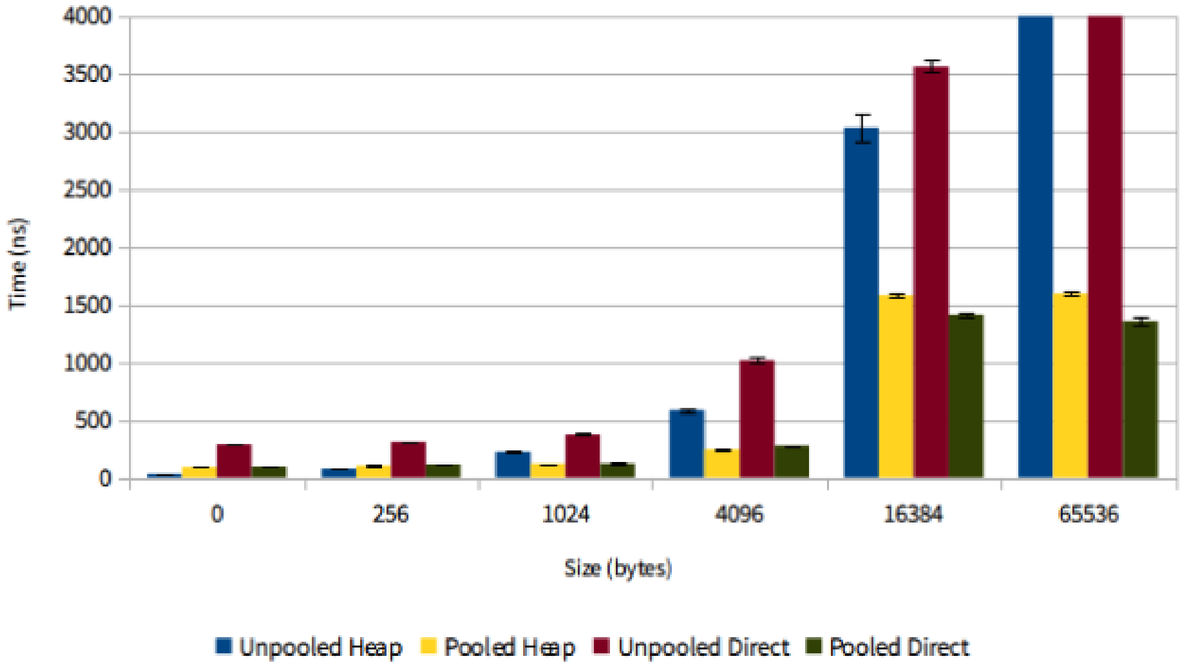
Netty 4: Use direct buffers by default and may even pool them.

Not depend on the GC for direct buffers.

https://www.flickr.com/photos/stavos52093/13807616194/

Use Pooling of buffers to reduce allocation / deallocation time!

- Pooling pays off for direct and heap buffers!



https://blog.twitter.com/2013/netty-4-at-twitter-reduced-gc-overhead

Issues with using non pooled-buffers

- Use unpooled buffers with caution!

Allocation / Deallocation is slow
Free up direct buffers == PITA!

Use pooled buffers!

```
Bootstrap bootstrap = new Bootstrap();
bootstrap.option(ChannelOption.ALLOCATOR, PooledByteBufAllocator.DEFAULT);
ServerBootstrap bootstrap = new ServerBootstrap();
bootstrap.childOption(ChannelOption.ALLOCATOR, PooledByteBufAllocator.DEFAULT);
```

Reference Counting - Wait What ?

⚠ Netty 4 uses reference counting for maximal performance!

```
public interface ReferenceCounted {
  int refCnt();
  ReferenceCounted retain();
  ReferenceCounted retain(int increment);
  boolean release();
  boolean release(int decrement);
}
```

Reference Counting - Who is responsible to release ?

- Rule of thumb is that the party who accesses a reference-counted object lastly is responsible for the destruction of the reference-counted object

https://www.flickr.com/photos/puppiesofpurgatory/3898011217/

- Important to understand who is responsible for release resources.

Reference Counting - ChannelInboundHandler(Adapter) ?

```
public class MyChannelInboundHandler extends ChannelInboundHandlerAdapter {
  public void channelRead(ChannelHandlerContext ctx, Object msg) {
    try {
      ...
    } finally {
      ReferenceCountUtil.release(msg);
    }
  }
}
```

You can also use `SimpleChannelInboundHandler` which calls `ReferenceCountUtil.release(msg)` for all messages it handles.

Reference Counting - ChannelOutboundHandler(Adapter) ?

You only want to call `ReferenceCountUtil.release(msg)` here if you don't call `ctx.write(originalMsg, promise)`.

Netty will automatically call `ReferenceCountUtil.release(msg)` once the transport has handled the outbound messages after they are flushed.

Reference Counting - Find the leak ?

simple leak reporting

```
LEAK: ByteBuf.release() was not called before it's garbage-collected....
```

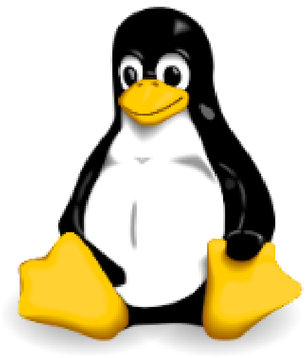
advanced leak reporting

```
LEAK: ByteBuf.release() was not called before it's garbage-collected.
Recent access records: 1
#1:
io.netty.buffer.AdvancedLeakAwareByteBuf.toString(AdvancedLeakAwareByteBuf.java:697)
    ...
Created at:
    ...

io.netty.handler.codec.xml.XmlFrameDecoderTest.testDecodeWithXml(XmlFrameDecoderTest.ja
```

Native stuff in Netty 4

- OpenSSL based SslEngine to reduce memory usage and latency.
- Native transport for Linux using Epoll ET for more performance and less CPU usage.
- Native transport also supports SO_REUSEPORT and TCP_CORK :)



Switching to native transport is easy

Using NIO transport

```
Bootstrap bootstrap = new Bootstrap().group(new NioEventLoopGroup());
bootstrap.channel(NioSocketChannel.class);
```

Using native transport

```
Bootstrap bootstrap = new Bootstrap().group(new EpollEventLoopGroup());
bootstrap.channel(EpollSocketChannel.class);
```

HTTP - Use Netty as HTTP server

https://www.flickr.com/photos/nadya/251716318

Related Codecs

- HTTP 1.0 / 1.1 and 2.0 (in review)
- HTTP Compression, CORS
- SPDY 3.1
- WebSockets and WebSockets Compression (in review)

Allow sending of HTTP responses / requests in chunks

→ 1 x `HttpResponse`, 0 - n `HTTPContent`, 1 x `LastHttpContent`

→ 1 x `HttpRequest`, 0 - n `HTTPContent`, 1 x `LastHttpContent`

Allows efficent streaming of HTTP responses / requests without big memory overhead.

Aggregate HTTP response / request parts

```
ChannelPipeline pipeline = channel.pipeline();
pipeline.addLast(new HttpObjectAggregator(10 * 1024 * 1024)); (1)
pipeline.addLast(new SimpleChannelInboundHandler<FullHttpRequest>() {
  @Override
  public void channelRead(ChannelHandlerContext ctx, FullHttpRequest req) { (2)
    // handle me
  }
});
```

(1) Add `HttpObjectAggregator` to `ChannelPipeline` which will take care of aggregating HTTP parts to `FullHttpResponse` or `FullHttpRequest` (incoming).

(2) Will only receive `FullHttpRequest` and so contains all parts for the request which includes headers, content and trailing headers.

Static header names and values via AsciiString.

```
private static final AsciiString X_HEADER_NAME = new AsciiString("X-Header"); ❶
private static final AsciiString X_VALUE = new AsciiString("Value");

pipeline.addLast(new SimpleChannelInboundHandler<FullHttpRequest>() {
  @Override
  public void channelRead(ChannelHandlerContext ctx, FullHttpRequest req) {
    FullHttpResponse response = new FullHttpResponse(HTTP_1_1, OK);
    response.headers().set(X_HEADER_NAME, X_VALUE); ❷
    ...
    ctx.writeAndFlush(response);
  }
});
```

❶ Create `AsciiString` for often used header names and values.

❷ Add to `HttpHeader` of `FullHttpResponse`

`AsciiString` is faster to encode and faster to find in `HttpHeaders`.

FileRegion for zero-memory-copy transfer (sendfile)

```
RandomAccessFile raf = new RandomAccessFile(file, "r");
HttpResponse response = new FullHttpResponse(HTTP_1_1, OK);
HttpHeaderUtil.setContentLength(response, raf.length());
channel.write(response); (1)
channel.write(new DefaultFileRegion(raf.getChannel(), 0, fileLength)); (2)
channel.writeAndFlush(LastHttpContent.EMPTY_LAST_CONTENT); (3)
```

(1) Write `HttpResponse` which contains the `HttpHeaders` with the Content-Length of the file set.

(2) Write a `DefaultFileRegion` which allows to use zero-memory-copy (transfer directly in kernel-space)

(3) Write a `LastHttpContent` to mark the response as complete.

You can only use `FileRegion` if data MUST NOT be converted on the fly.

HttpChunkInput when FileRegion can not be used

```
File f = new File(file, "r");
HttpResponse response = new FullHttpResponse(HTTP_1_1, OK);
HttpHeaders.setContentLength(response, f.length());
channel.write(response); (1)
channel.write(new HttpChunkedInput(new ChunkedNioFile(file))); (2)
channel.writeAndFlush(LastHttpContent.EMPTY_LAST_CONTENT); (3)
```

(1) Write `HttpResponse` which contains the `HttpHeaders` with the Content-Length of the file set.

(2) Write a `HttpChunkedInput` to stream from a file when FileRegion can't be used

(3) Write a `LastHttpContent` to mark the response as complete.

`ChunkedWriteHandler` must be added to the `ChannelPipeline` for this to work!

Use this when you have for example the `SslHandler` in the `ChannelPipeline`.

Validate headers or not ?

Validate for headers for US-ASCII

```
ChannelPipeline pipeline = channel.pipeline();
pipeline.addLast(new HttpRequestDecoder(4096, 8192, 8192));

HttpResponse response = new DefaultHttpResponse(HttpVersion.HTTP_1_1, HttpStatus.OK);
```

Not validate for headers for US-ASCII

```
ChannelPipeline pipeline = channel.pipeline();
pipeline.addLast(new HttpRequestDecoder(4096, 8192, 8192, false));

HttpResponse response = new DefaultHttpResponse(HttpVersion.HTTP_1_1, HttpStatus.OK,
false);
```

Validation takes time and most of the times it is not needed directly in the decoder/encoder.

Easy HTTP Pipelining to save syscalls!

```
public class HttpPipeliningHandler extends SimpleChannelInboundHandler<HttpRequest> {
  @Override
  public void channelRead(ChannelHandlerContext ctx, HttpRequest req) {
    ChannelFuture future = ctx.writeAnd(createResponse(req)); ❶
    if (!HttpHeaders.isKeepAlive(req)) {
      future.addListener(ChannelFutureListener.CLOSE); ❷
    }
  }
  @Override
  public void channelReadComplete(ChannelHandlerContext ctx) {
    ctx.flush(); ❸
  }
}
```

❶ Write to the `Channel` (No syscall!) but don't flush yet

❷ Close socket when done writing

❸ Flush out to the socket.

HTTP Server that needs to pipe to external services

```
public class HttpHandler extends SimpleChannelInboundHandler<FullHttpRequest> {
  @Override
  public void channelRead0(ChannelHandlerContext ctx, FullHttpRequest) { ❶
    final Channel inboundChannel = ctx.channel();
    Bootstrap b = new Bootstrap();
    b.group(new NioEventLooopGroup()); ❷
    ...
    ChannelFuture f = b.connect(remoteHost, remotePort);
    ...
  }
```

❶ Called once a new `FullHttpRequest` is received

❷ Use a new `EventLoopGroup` instance to handle the connection to the remote peer

Don’t do this! This will tie up more resources than needed and introduce extra context-switching overhead.

HTTP Server that needs to pipe to external services which reduce context-switching to minimum

```
public class HttpHandler extends SimpleChannelInboundHandler<FullHttpRequest> {
  @Override
  public void channelRead0(ChannelHandlerContext ctx, FullHttpRequest) { (1)
    final Channel inboundChannel = ctx.channel();
    Bootstrap b = new Bootstrap();
    b.group(inboundChannel.eventLoop()); (2)
    ...
    ChannelFuture f = b.connect(remoteHost, remotePort);
    ...
  }
}
```

(1) Called once a new connection is accepted

(2) Share the same `EventLoop` between both Channels. This means all IO for both connected Channels are handled by the same Thread.

Always share EventLoop in those Applications

By default Netty will keep on reading data from the `Channel` once something is ready.

`Need more fine grained control ?`

```
channel.config().setAutoRead(false); (1)
channel.read(); (2)
channel.config().setAutoRead(true); (3)
```

(1) Disable auto read == no more data will be read automatically from this `Channel`.

(2) Tell the `Channel` to do one read operation once new data is ready

(3) Enable again auto read == Netty will automatically read again

This can also be quite useful when writing proxy like applications!

HTTP Server makes use of AutoRead

```
public class HttpHandler extends SimpleChannelInboundHandler<FullHttpRequest> {
  @Override public void channelRead0(final ChannelHandlerContext ctx, FullHttpRequest)
{
    ctx.channel().setAutoRead(false); ❶
    Bootstrap b = createBootstrap();
    b.connect(remoteHost, remotPort).addListener(new ChannelFutureListener() { ❷
      public void operationComplete(ChannelFuture future) {
        if (future.isSuccess()) ctx.channel().setAutoRead(true); ❸
      }
    });
  }
}
```

❶ Stop reading from the inbound `Channel`

❷ Connect to remote host and add `ChannelFutureListener`

❸ Start reading from inbound `Channel` again once done

Disable reading helps with memory and backpressure!

Use HttpHeaders static utility methods

- `HttpHeaders.Names` and `HttpHeaders.Values` contain static final fields that should be used as these are optimized for the encoder.

- `HttpHeaders` contains static methods to act on the headers which can often use a fast-path because of implementation details.

Don’t send more headers than needed

`HTTP/1.1` uses keep-alive by default, so no need to send keep-alive header.

Removing a header is an improvement in terms of speed as you save the encoding and also the bandwidth to transfer it.

File transfer ?

Use zero-memory-copy for efficient transfer of raw file content

```
Channel channel = ...;
FileChannel fc = ...;
channel.writeAndFlush(new DefaultFileRegion(fc, 0, fileLength));
```

This only works if you don't need to modify the data on the fly. If so use `ChunkedWriteHandler` and `NioChunkedFile`.

Add support for HttpCompression

https://www.flickr.com/photos/marcovdz/4520986339/

```
pipeline.addLast(new HttpContentCompressor()); (1)
```

(1) Add compression support to HTTP Server. Supports DEFLATE and GZIP :)

Add support for WebSockets

```
pipeline.addLast(new WebSocketServerProtocolHandler("/ws")); (1)
pipeline.addLast(new SimpleChannelInboundHandler<TextWebSocketFrame>() { (2)
  @Override
  public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {
    ctx.writeAndFlush(msg);
  }
});
```

(1) Add support for WebSockets on path /ws

(2) ChannelInboundHandler that will echo back any TextWebSocketFrame

Add support for SPDY

```
public class SpdyServerInitializer extends ChannelInitializer<SocketChannel> {
    private final SslContext sslCtx =  ....
    @Override
    public void initChannel(SocketChannel ch) {
        ChannelPipeline p = ch.pipeline();
        p.addLast(sslCtx.newHandler(ch.alloc()));
        p.addLast(new SpdyOrHttpChooser() { (1)
          @Override
          protected ChannelInboundHandler createHttpRequestHandlerForHttp() { ... } (2)
          @Override
          protected ChannelInboundHandler createHttpRequestHandlerForSpdy() { ... } (3)
        });
    }
}
```

(1) Add SpdyOrHttpChooser which will detect if SPDY or HTTP should be used

(2) ChannelInboundHandler that will handle HttpRequests which are done via HTTP

What we learned while working on Netty

https://www.flickr.com/photos/21847073@N05/5850264509/

GC-Pressure

Creating a lot of objects has a very bad impact on GC times and so throughput/latency when you push hard enough.

Think about Objects and how you can create less or share immutable ones.

Memory usage

- Save memory is important in long-living objects like our Channel implementations as there may be 100k's of them active at the same time.

- Small changes here can have a big impact.

- By replacing `AtomicReference with `AtomicReferenceFieldUpdater` we were able to save ca. 3GB heap for a user with 500k concurrent connections.

Direct ByteBuffer reclaimation by GC doesn't work fast enough

Depending on the GC, reclaiming memory of Direct `ByteBuffer` just doesn't work fast enough and may not be reclaimed before you see an OOME.

User `sun.misc.Cleaner` to release memory ASAP. No risk not fun…

```
static void freeDirectBuffer(ByteBuffer buffer) {
  if (CLEANER_FIELD_OFFSET == -1 || !buffer.isDirect()) {
    return;
  }
  try {
    Cleaner cleaner = (Cleaner) getObject(buffer, CLEANER_FIELD_OFFSET);
    if (cleaner != null) cleaner.clean();

  } catch (Throwable t) { // Nothing we can do here. }
}
```

Always use direct ByteBuffer when writing to SocketChannel

If you don't use a direct `ByteBuffer` the OpenJDK / Oracle Java will do an extra memory copy to transfer the data direct `ByteBuffer`.

Implementing your own pool can be a big win here like what we now have with `PooledByteBufAllocator`

Range checks can be expensive - ByteBufProcessor to the rescue…

Range checks can be quite expensive, minimize these.

SlowSearch for ByteBuf :(

```
int index = -1;
for (int i = buf.readerIndex(); index == -1 && i <  buf.writerIndex(); i++) {
  if (buf.getByte(i) == '\n') {
    index = i;
  }
}
```

FastSearch for ByteBuf :)

```
int index = buf.forEachByte(new ByteBufProcessor() {
  @Override
  public boolean process(byte value) {
    return value != '\n';
  }
});
```

Commit-Template for the win!

Using a commit-template makes it easy for team members to understand your changes.

Correctly return from selector loop one a scheduled task is ready for... ···

016bdfbf66

Browse code

```
… processing

Motivation:

We use the nanoTime of the scheduledTasks to calculate the milli-seconds to wait for a select operation
to select something. Once these elapsed we check if there was something selected or some task is ready
for processing. Unfortunally we not take into account scheduled tasks here so the selection loop will
continue if only scheduled tasks are ready for processing. This will delay the execution of these tasks.

Modification:

- Check if a scheduled task is ready after selecting
- also make a tiny change in NioEventLoop to not trigger a rebuild if nothing was selected because the
timeout was reached a few times in a row.

Result:

Execute scheduled tasks on time.
```

normanmaurer authored 2 days ago

Checkstyle and review for good code-quality.

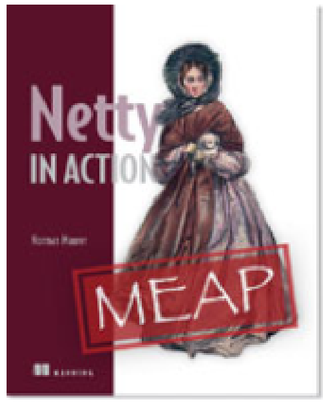
- Using checkstyle rules during build to fail on inconsistent code styling
- For more complex changes use review processes before merging in changes
- Work with git branches
- Use `git rebase` and `git cherry-pick` to keep history clean

Semantic Versioning - for the win

Use Semantic Versioning to make it easy for your users to upgrade etc.

Want to know more?

Buy my book Netty in Action and make me RICH.



http://www.manning.com/maurer

$ KA-CHING $

References

- Netty - http://netty.io
- Slides generated with Asciidoctor and DZSlides backend
- Original slide template - Dan Allen & Sarah White

Norman Maurer

@normanmaurer