Netty - Best Practices a.k.a Faster == Better

@Twitter 2014 · San Francisco · 2014/07/28

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

- Netty / All things NIO
- Author of Netty in Action
- @normanmaurer
- github.com/normanmaurer

Agenda

- Pipelining
- Writing gracefully
- Buffers best-practises
- EventLoop

No Pipelining Optimization

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

❶ Write to the Channel and flush out to the Socket.

❷ After written close Socket

Pipelining to safe syscalls!

```java
public class HttpPipeliningHandler extends SimpleChannelInboundHandler<HttpRequest> {
  @Override
  public void channelRead(ChannelHandlerContext ctx, HttpRequest req) {
    ChannelFuture future = ctx.writeAnd(createResponse(req)); ❶
    if (!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.

write(msg) , flush() and writeAndFlush(msg)

write(msg) ⇒ pass through pipeline

flush() ⇒ gathering write of previous written msgs

writeAndFlush() ⇒ short-cut for write(msg) and flush()

- Limit flushes as much as possible as syscalls are quite expensive.

- But also limit `write(...)` as much as possible as it need to traverse the whole pipeline.

May write too fast!

```java
public class BlindlyWriteHandler extends ChannelInboundHandlerAdapter {
  @Override
  public void channelActive(ChannelHandlerContext ctx) throws Exception {
    while(needsToWrite) { ❶
        ctx.writeAndFlush(createMessage());
    }
  }
}
```

❶ Writes till `needsToWrite` returns `false`.

Risk of OutOfMemoryError if writing too fast and having slow receiver!

Correctly write with respect to slow receivers

```
public class GracefulWriteHandler extends ChannelInboundHandlerAdapter {
  @Override
  public void channelActive(ChannelHandlerContext ctx) {
    writeIfPossible(ctx.channel());
  }
  @Override
  public void channelWritabilityChanged(ChannelHandlerContext ctx) {
    writeIfPossible(ctx.channel());
  }

  private void writeIfPossible(Channel channel) {
    while(needsToWrite && channel.isWritable()) { ❶
      channel.writeAndFlush(createMessage());
    }
  }
}
```

❶ Make proper use of `Channel.isWritable()` to prevent OutOfMemoryError

Configure high and low write watermarks

Set sane WRITE_BUFFER_HIGH_WATER_MARK and WRITE_BUFFER_LOW_WATER_MARK

Server

```
ServerBootstrap bootstrap = new ServerBootstrap();
bootstrap.childOption(ChannelOption.WRITE_BUFFER_HIGH_WATER_MARK, 32 * 1024);
bootstrap.childOption(ChannelOption.WRITE_BUFFER_LOW_WATER_MARK, 8 * 1024);
```

Client

```
Bootstrap bootstrap = new Bootstrap();
bootstrap.option(ChannelOption.WRITE_BUFFER_HIGH_WATER_MARK, 32 * 1024);
bootstrap.option(ChannelOption.WRITE_BUFFER_LOW_WATER_MARK, 8 * 1024);
```

Issues with using non pooled buffers

Use unpooled buffers with caution!

Allocation / Deallocation is slow

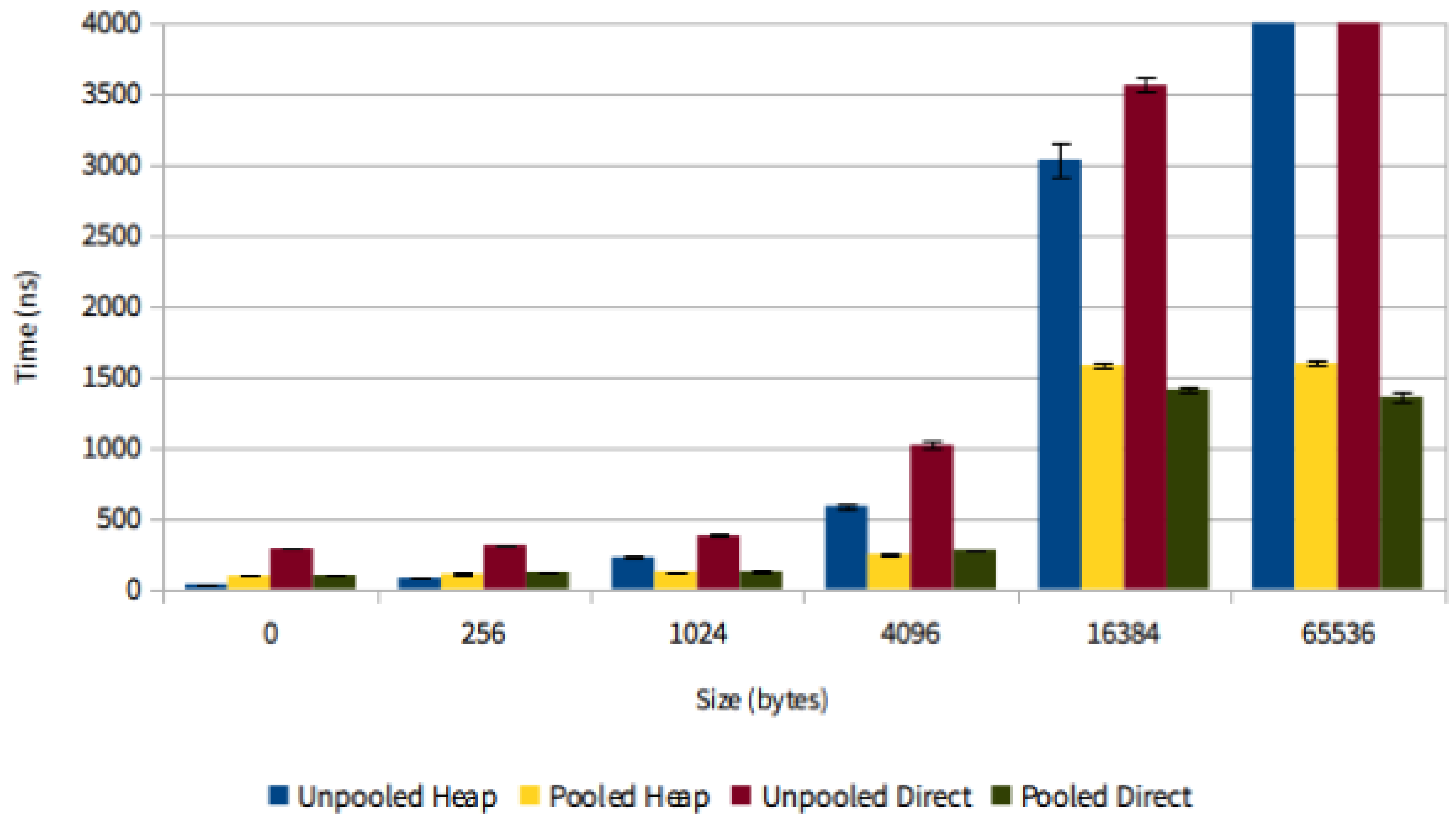
Free up direct buffers == PITA!

Use pooled buffers!

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

Use Pooling of buffers to reduce allocation / deallocation time!

- Pooling pays off for direct and heap buffers!



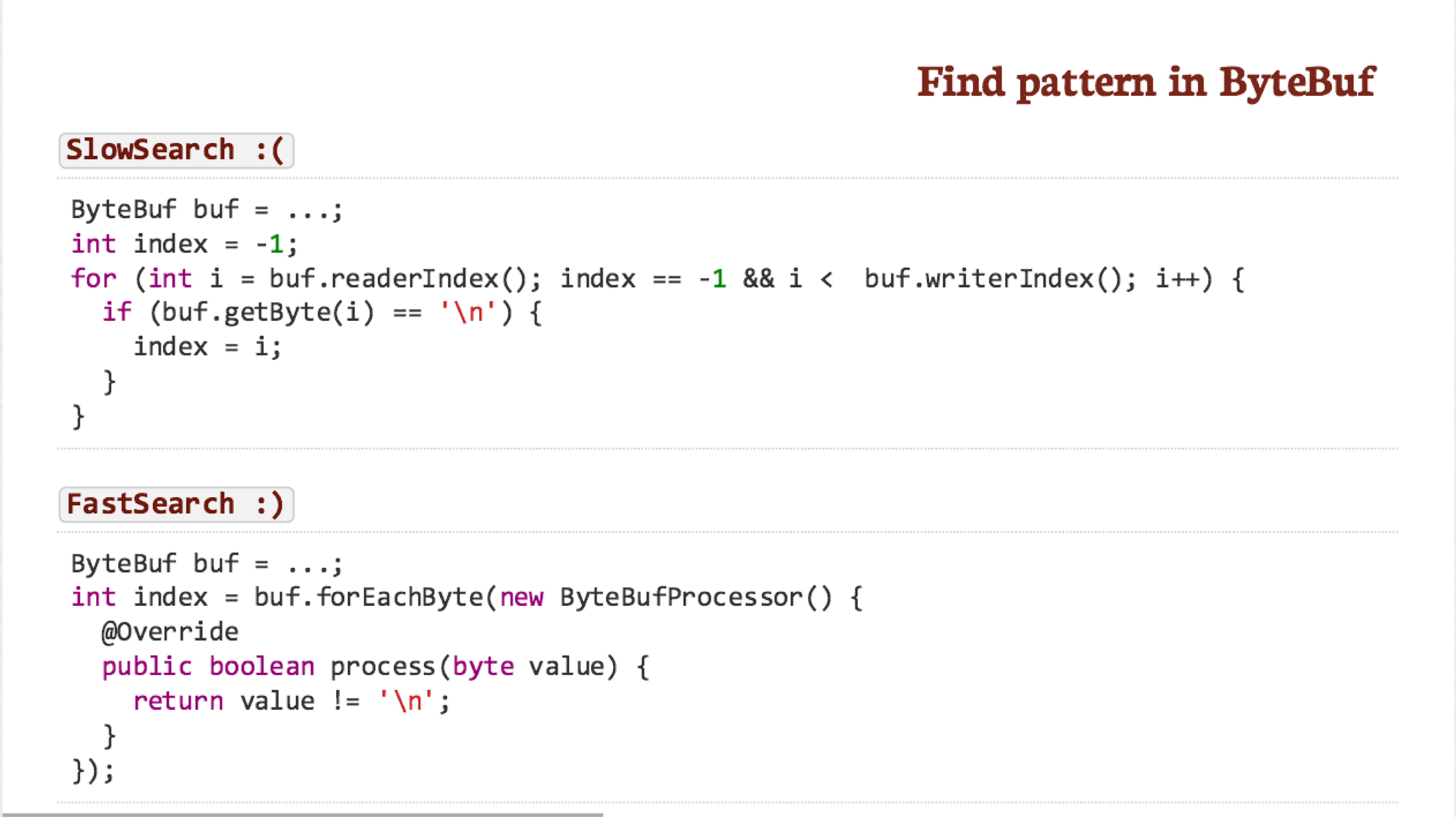
https://blog.twitter.com/2013/netty-4-at-twitter-reduced-gc-overhead

Always use direct ByteBuffer when writing to SocketChannel

- OpenJDK and Oracle JDK copy otherwise to direct buffer by itself!

Only use heap buffers if need to operate on byte[]` in `ChannelOutboundHandler`! By default direct ByteBuf` will be returned by `ByteBufAllocator.buffer(...)`.

Take this as rule of thumb



Find pattern in ByteBuf

SlowSearch :(

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

FastSearch :)

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

Messages with Payload? Yes please…

`ByteBuf` **payload ⇒ extend** `DefaultByteBufHolder`

- ℹ️ reference-counting for free
- ℹ️ release resources out-of-the-box

http://www.flickr.com/photos/za3tooor/65911648/

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`.

Never block the EventLoop!

- `Thread.sleep()`
- `CountDownLatch.await()` or any other blocking operation from `java.util.concurrent`
- Long-lived computationally intensive operations
- Blocking operations that might take a while (e.g. DB query)

http://www.flickr.com/photos/za3tooor/65911648/

Re-use EventLoopGroup if you can!

```java
Bootstrap bootstrap = new Bootstrap().group(new NioEventLoopGroup());
Bootstrap bootstrap2 = new Bootstrap().group(new NioEventLoopGroup());
```

`Share EventLoopGroup between different Bootstraps`

```java
EventLoopGroup group = new NioEventLoopGroup();
Bootstrap bootstrap = new Bootstrap().group(group);
Bootstrap bootstrap2 = new Bootstrap().group(group);
```

Sharing the same `EventLoopGroup` allows to keep the resource usage (like Thread-usage) to a minimum.

Proxy like application with context-switching issue

```
public class ProxyHandler extends ChannelInboundHandlerAdapter {
  @Override
  public void channelActive(ChannelHandlerContext ctx) { 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 connection was accepted

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.

Proxy like application which reduce context-switching to minimum

```
public class ProxyHandler extends ChannelInboundHandlerAdapter {
  @Override
  public void channelActive(ChannelHandlerContext ctx) { ❶
    final Channel inboundChannel = ctx.channel();
    Bootstrap b = new Bootstrap();
    b.group(inboundChannel.eventLoop()); ❷
    ...
    ChannelFuture f = b.connect(remoteHost, remotePort);
    ...
  }
}
```

❶ Called once a new connection was accepted

❷ 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

Operations from inside ChannelHandler

```
public class YourHandler extends ChannelInboundHandlerAdapter {
  @Override
  public void channelActive(ChannelHandlerContext ctx) {
    // BAD (most of the times)
    ctx.channel().writeAndFlush(msg); ❶

    // GOOD
    ctx.writeAndFlush(msg); ❷
   }
}
```

❶ `Channel.*` methods ⇒ the operation will start at the tail of the `ChannelPipeline`

❷ `ChannelHandlerContext.* methods => the operation will start from this `ChannelHandler` to flow through the `ChannelPipeline`.

💡 Use the shortest path as possible to get the maximal performance.

Share ChannelHandlers if stateless

```
@ChannelHandler.Shareable ❶
public class StatelessHandler extends ChannelInboundHandlerAdapter {
  @Override
  public void channelActive(ChannelHandlerContext ctx) {
    logger.debug("Now client from " + ctx.channel().remoteAddress().toString());
   }
}

public class MyInitializer extends ChannelInitializer<Channel> {
  private static final ChannelHandler INSTANCE = new StatelessHandler();
  @Override
  public void initChannel(Channel ch) {
    ch.pipeline().addLast(INSTANCE);
  }
}
```

❶ Annotate `ChannelHandler` that are stateless with `@ChannelHandler.Shareable` and use the same instance accross Channels to reduce GC.

Remove ChannelHandler once not needed anymore

```java
public class OneTimeHandler extends ChannelInboundHandlerAdapter {
  @Override
  public void channelActive(ChannelHandlerContext ctx) {
    doOneTimeAction();
    ctx.channel().pipeline().remove(this); ❶
   }
}
```

❶ Remove `ChannelHandler` once not needed anymore.

This keeps the `ChannelPipeline` as short as possible and so eliminate overhead of traversing as much as possible.

Use proper buffer type in MessageToByteEncoder

```
public class EncodeActsOnByteArray extends MessageToByteEncoder<YourMessage> {
  public EncodeActsOnByteArray() { super(false); } ❶
  @Override
  public encode(ChannelHandlerContext ctx, YourMessage msg, ByteBuf out) {
    byte[] array = out.array(); ❷
    int offset = out.arrayOffset() + out.writerIndex();
    out.writeIndex(out.writerIndex() + encode(msg, array, offset)); ❸
  }
  private int encode(YourMessage msg, byte[] array, int offset, int len) { ... }
}
```

❶ Ensure heap buffers are used when pass into `encode(...)` method. This way you can access the backing array directly

❷ Access the backing array and also calculate offset

❸ Update writerIndex to reflect written bytes

This saves extra byte copies.

To auto-read or not to auto-read

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

To auto-read or not to auto-read

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

`Need more fine grained control ?`

```
channel.config().setAutoRead(false); ❶
channel.read(); ❷
channel.config().setAutoRead(true); ❸
```

❶ Disable auto read == no more data will be read automatically from this `Channel`.

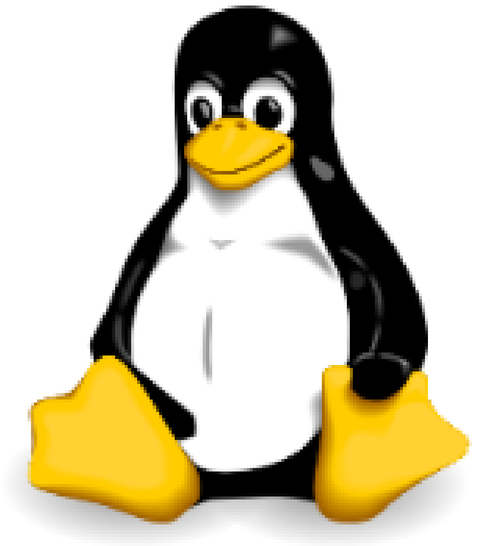
❷ Tell the `Channel` to do one read operation once new data is ready

❸ Enable again auto read == Netty will automatically read again

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

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

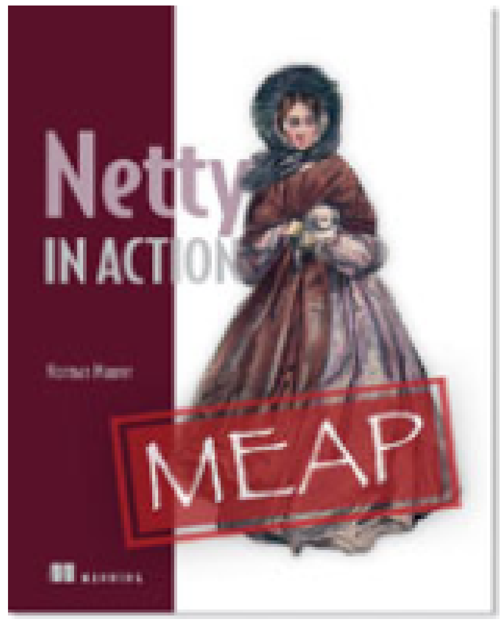
```
Bootstrap bootstrap = new Bootstrap().group(new NioEventLoopGroup());
bootstrap.channel(NioSocketChannel.class);
```

Using native transport

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

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
- All pictures licensed with `Creative Commons Attribution` or `Creative Commons Attribution-Share Alike`

Norman Maurer

🐦 @normanmaurer