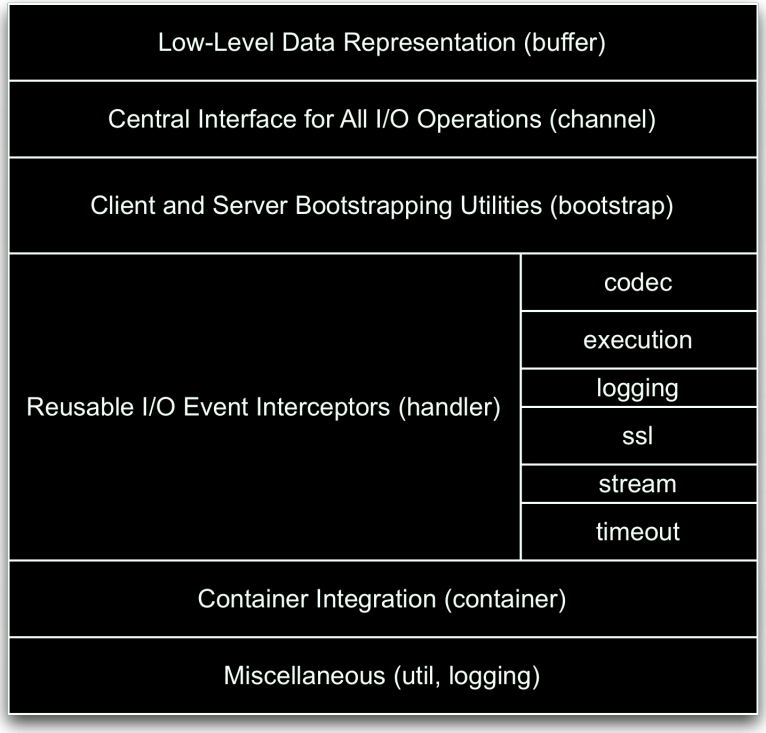
NETTY 3.1.0CR

Michael McGrady



INDEX

Chapter 1
BUFFER

Abstraction of a byte buffer, the fundamental data structure to represent low-level binary and text messages.

org.jboss.netty.buffer

Netty uses its own buffer API instead of NIO (java.nio.ByteBuffer) to represent a sequence of bytes. The objective had been to cure the problems with the NIO buffer.

1.1. Features

Features of the Netty buffer include:

- Extensibility: You can define a custom buffer type.
- Transparent Zero Copy: A transparent zero copy is achieved by a built-in composite buffer type.
- Automatic Capacity Extension: A dynamic buffer type is provided out of the box that has an on demand expandable capacity like `java.lang.StringBuffer`.
- There is no need to call the flip method anymore.
- Better Performance: The Netty buffer is often faster than the Java buffer.

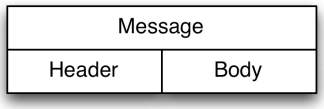
1.1.1. Extensibility

The Netty buffer (`org.jboss.netty.buffer.ChannelBuffer`) has a rich set of operations optimized for rapid protocol implementation. The buffer, for example, provides various operations for accessing unsigned values and strings and for searching for certain byte sequence in a buffer. You can also wrap an existing buffer type to add convenient accessors while still implementing the `ChannelBuffer` type.

1.1.2. Transparent Zero Copy

To lift performance to an extreme you need to reduce the number of the memory copy operation. You might have a set of buffers that could be sliced and combined to compose a whole new message. Netty provides a composite buffer that allows you to create a new buffer from an arbitrary number of existing buffers with no memory copy.

For example, a header and a body could be produced by separate operations and assembled when a message is sent.



If a `java.nio.ByteBuffer` were used, you would have to create a new big byte buffer and copy the two parts into the new buffer. Or, you might perform a gathering write operation in NIO, but this would restrict you to representing the composite of buffers as an array of ByteBuffers rather than as a single buffer, breaking the abstraction and introducing complicated state management. Also,

this would be of no use if you were not going to write from a NIO channel. The composite type is incompatible with the component type:

```
ByteBuffer [] message = new ByteBuffer {} {header,body} ;
```

`ChannelBuffer` does not have such caveats because it is fully extensible and has a built-in composite buffer type, viz. `CompositeChannelBuffer`. This composite type is compatible with the component type:

```
ChannelBuffer message = ChannelBuffers.wrappedBuffer
(header,body);
```

Therefore, you can create a composite by mixing a composite with an ordinary buffer.

```
ChannelBuffer messageWithFooter =
ChannelBuffers.wrappedBuffer (message,footer);
```

Because the composite is still a `ChannelBuffer`, it will behave just like a single buffer. The unsigned integer being read in the following code is located across body and footer:

```
messageWithFooter.readableBytes() –
footer.readableBytes() – 1 ;
```

1.1.3. Automatic Capacity Extension

Many protocols define variable length messages, which means that you cannot determine the length until you construct the message. Netty allows this, like with `StringBuffer`, with a dynamic buffer created by the following method:

```
org.jboss.netty.buffer.ChannelBuffers.dynamicBuffer()
```

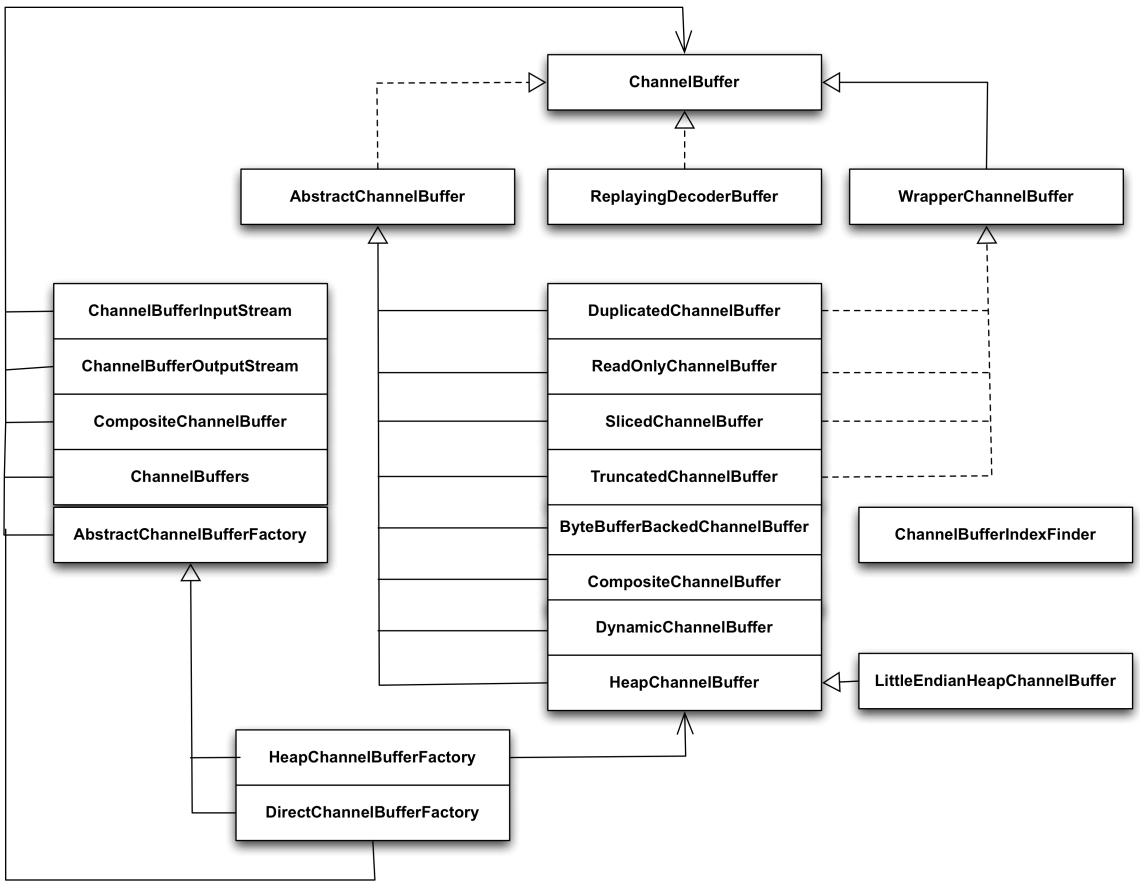
1.1.4. Better Performance

The most frequently used buffer implementation of `ChannelBuffer` is a thin wrapper of a byte array. Unlike `ByteBuffer` it has no complicated boundary check and index compensation, so it is easier for a JVM to optimize the buffer access. More complicated buffer implementation is used only for sliced (`SlicedChannelBuffer`) or composite (`CompositeChannelBuffer`) buffers, and these perform as well as `ByteBuffer`.

1.2. Interfaces

1.2.1. `ChannelBuffer`

A random and sequentially accessible sequence of zero or more bytes (octets). This interface provides an abstract view of one or more primitive byte arrays (`byte [ ]`) and NIO buffers (`ByteBuffer`).



1.2.1.1. Creation of a Buffer

Creating a buffer by using the helper methods in `ChannelBuffer`s is recommended as opposed to calling an individual implementation's constructor.

`HeapChannelBufferFactory` is used to create a `HeapChannelBuffer`, which relies upon the JVM garbage collector that is highly optimized for heap allocation. `DirectChannelBufferFactory` pre-allocates a large chunk of direct buffer and returns its slice on demand. Direct buffers are reclaimed via `ReferenceQueue` in most JDK implementations and are de-allocated less efficiently than an ordinary heap buffer. `OutOfMemoryError` problems will result when allocating small direct buffers more often than the GC throughput of direct buffers, which is much lower than the GC throughput of heap buffers. This factory avoids this problem by allocating a large chunk of pre-allocated direct buffer and reducing the number of the garbage collected internal direct buffer objects.

1.2.1.2. Random Access Indexing

Just like an ordinary byte array, `ChannelBuffer` uses an array element http://en.wikipedia.org/wiki/Index_(information_technology). This means the first

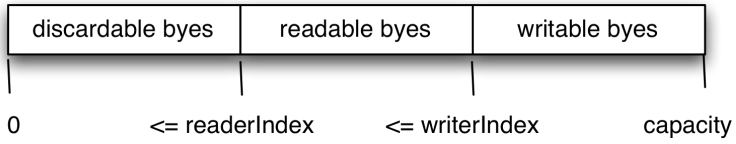
byte is always 0 and the last byte s always capacity – 1. To iterate all the bytes of a buffer, you can do the following, regardless of the internal implementation:

```
ChannelBuffer buffer = ...;

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

1.2.1.3. Sequential Access Indexing

`ChannelBuffer` provides two pointer variables to support sequential read and write operations: (a) `readerIndex` for a read and (b) `writerIndex` for a write. The following diagram shows how a buffer is segmented into three areas by the two pointers.



1.2.1.4. Readable Bytes (Actual Content)

This is the actual data. Operations whose names start with "`read`" or "`skip`" will get or skip the data at the current `readerIndex` and increase it by the number of read bytes. If the argument of the read operation is a `ChannelBuffer` and no destination index is specified, the specified buffer's `readerIndex` is increased together. If there is not enough content left, an `IndexOutOfBoundsException` is raised. The default value of a newly allocated, wrapped or copied buffer's `readerIndex` is 0. The following code iterates the readable bytes of a buffer.

```
ChannelBuffer buffer = ...;
while(buffer.readable()) {
   System.out.println(buffer.readByte());
}
```

1.2.1.5. Writable Byes

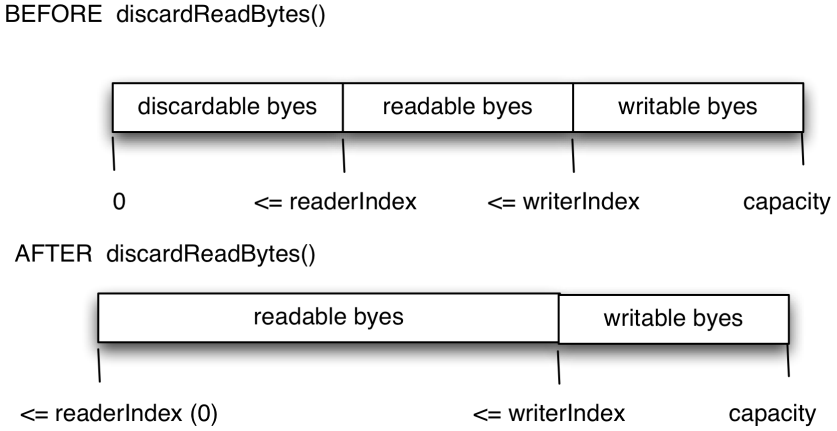
This segment is an undefined space which needs to be filled. Any operation whose name ends with "`write`" will write the data at the current writerIndex and increase it by the number of written bytes. If the argument is a `ChannelBuffer`, and no source index is specified, the specified buffer's `readerIndex`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

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

The default value of wrapped or copied buffer's `writerIndex` is the capacity of the buffer.

1.2.1.5. Discardable Bytes

These bytes were already read by a read operation. Read bytes can be discarded by calling `discardReadBytes` to reclaim unused area as depicted in the following diagram.



1.2.1.6. Search Operations

Netty provides various `ChannelBuffer indexOf` methods for searching for the index of a value that meets certain criteria. Complicated dynamic sequential searches can be done with `ChannelBufferIndexFinder` as well as static single byte search.

1.2.1.7. Mark and Reset

You can reposition `readerIndex` and `writerIndex` by calling a reset method. This works similar to he mark and reset methods in `InputStream` except that there is no `readLimit`.

1.2.1.8. Derived Buffers

You can create a view of an existing buffer with `duplicate` and `slice` or `slice(int, int)`. A derived buffer will have an independent `readerIndex` and `writerIndex`, while it shares other internal data representation, just like a NIO buffer.

1.2.1.9. NIO Buffers

Various `toByeBuffer` and `toByteBuffers` method convert a `ChannelBuffer` into one or more NIO buffers. These methods avoid buffer allocation and memory cop wherever possible, but there are no guarantees that none will occur.

1.2.1.10 Strings

Various `toString` methods convert a `ChannelBuffer` to a `String`. The method `toString` is not a conversion method.

1.2.1.11. Streams

Please refer to `ChannelInputStream` and `ChannelBufferOutputStream`.

Chapter 2
CHANNEL

The core channel API which is an asynchronous and event-driven abstraction of various transports such as an NIO Channel.	org.jboss.netty.channel
A channel registry which helps a user maintain a list of open Channels and perform bulk operations on them.	org.jboss.netty.channel.group
A virtual transport that enables the communication between two parties n the same virtual machine.	org.jboss.netty.channel.local
Abstract TCP and UDP socket interfaces which extend the core channel API	org.jboss.netty.channel.socket
An HTTP based client-side SocketChannel and is corresponding server-side Servlet implementation that make our existing server application work in a firewalled network.	org.jboss.netty.channel.socket.http
NIO based socket channel API implementation - recommended for a large number of connections (>=1000).	org.jboss.netty.channel.socket.nio
Old blocking I/O based socket channel API implementation - recommended for a small number of connections (<=1000)	org.jboss.netty.channel.socket.oio
An alternative channel API implementation which uses JBoss XNIO as its I/O provider	org.jboss.netty.channel.xnio

Chapter 3
BOOTSTRAP

IoC friendly helper classes which enable an easy implementation of typical client side and server side channel initialization.

org.jboss.netty.bootstrap

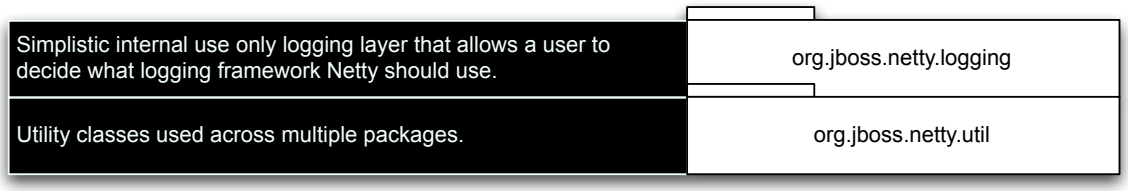
Chapter 4
HANDLER

Encoder and decoder which transform a Base64-encoded String or ChannelBuffer into a decoded ChannelBuffer and vice versa.	org.jboss.netty.handler.codec.base64
A helper that wraps an encoder or decoder so that they can be used without doing actual I/O in unit tests or higher level codecs.	org.jboss.netty.handler.codec.embedder
Extensible decoder and its common implementations which deal with the packet fragmentation and reassembly issue found in a stream-based transport such as TCP/IP.	org.jboss.netty.handler.codec.frame
Encoder, decoder and their related message types for HTTP.	org.jboss.netty.handler.codec.http
Simplistic abstract classes which help implement encoder and decoder that transform an object into another object and vice versa.	org.jboss.netty.handler.codec.oneone
Encoder and decoder that transform a Google Protocol Buffers Message into a ChannelBuffer and vice versa.	org.jboss.netty.handler.codec.protobuf
Specialized variation of FrameDecoder that enables implementation of a non-blocking decoder in the blocking I/O paradigm.	org.jboss.netty.handler.codec.replay
Encoder, decoder and their compatibility stream implementations that transform a Serializable object into a byte buffer and vice versa.	org.jboss.netty.handler.codec.serialization
Encoder, decoder and their compatibility stream implementations that transform a Serializable object into a byte buffer and vice versa.	org.jboss.netty.handler.codec.serialization
Encoder and decoder which transform a String into a ChannelBuffer and vice versa.	org.jboss.netty.handler.codec.string
Executor-based implementation of various thread models and memory overload prevention mechanisms.	org.jboss.netty.handler.execution
Logs a ChannelEvent for debugging purposes using an InternalLogger.	org.jboss.netty.handler.logging
SSL/TLS implementation based on SSLEngine	org.jboss.netty.handler.ssl
Writes very large data stream asynchronously neither spending a lot of memory nor getting OutOfMemoryError.	org.jboss.netty.handler.stream
Adds support for read and write timeout and idle connection notification using a Timer.	org.jboss.netty.handler.timeout

Chapter 5
CONTAINER

Chapter 6

MISCELLANEOUS



Chapter 7

EXAMPLES