第八章：附带的ChannelHandler和Codec

目录 [-]

本章介绍

- 使用SSL/TLS创建安全的Netty程序
- 使用Netty创建HTTP/HTTPS程序
- 处理空闲连接和超时
- 解码分隔符和基于长度的协议
- 写大数据
- 序列化数据

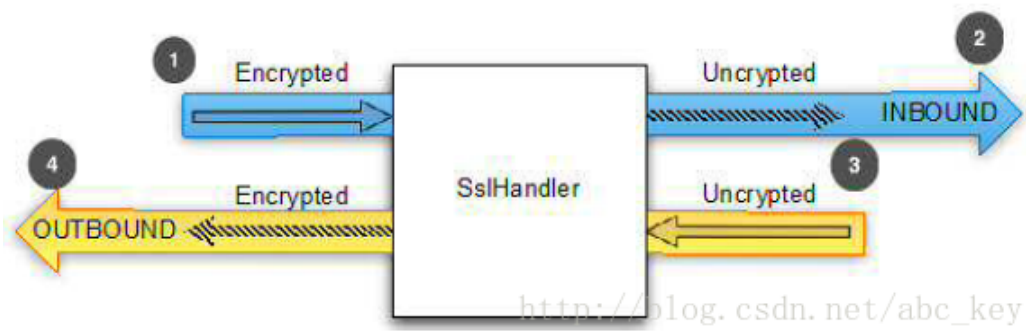
上一章讲解了如何创建自己的编解码器，我们现在可以用上一章的知识来编写自己的编解码器。不过Netty提供了一些标准的ChannelHandler和Codec。Netty提供了很多协议的支持，所以我们不必自己发明轮子。Netty提供的这些实现可以解决我们的大部分需求。本章讲解Netty中使用SSL/TLS编写安全的应用程序，编写HTTP协议服务器，以及使用如WebSocket或Google的SPDY协议来使HTTP服务获得更好的性能；这些都是很常见的应用，本章还会介绍数据压缩，在数据量比较大的时候，压缩数据是很有必要的。

8.1 使用SSL/TLS创建安全的Netty程序

通信数据在网络上传输一般是不安全的，因为传输的数据可以发送纯文本或二进制的数据，很容易被破解。我们很有必要对网络上的数据进行加密。SSL和TLS是众所周知的标准和分层的协议，它们可以确保数据时私有的。例如，使用HTTPS或SMTPS都使用了SSL/TLS对数据进行了加密。

对于SSL/TLS，Java中提供了抽象的SslContext和SslEngine。实际上，SslContext可以用来获取SslEngine来进行加密和解密。使用指定的加密技术是高度可配置的，但是这不在本章范围。Netty扩展了Java的SslEngine，添加了一些新功能，使其更适合基于Netty的应用程序。Netty提供的这个扩展是SslHandler，是SslEngine的包装类，用来对网络数据进行加密和解密。

下图显示SslHandler实现的数据流：



上图显示了如何使用ChannelInitializer将SslHandler添加到ChannelPipeline，看下面代码：

```java
public class SslChannelInitializer extends ChannelInitializer<Channel> {

    private final SSLContext context;
    private final boolean client;
    private final boolean startTls;

    public SslChannelInitializer(SSLContext context, boolean client, boolean startTls) {
        this.context = context;
        this.client = client;
```

```
        this.startTls = startTls;
    }

    @Override
    protected void initChannel(Channel ch) throws Exception {
        SSLEngine engine = context.createSSLEngine();
        engine.setUseClientMode(client);
        ch.pipeline().addFirst("ssl", new SslHandler(engine, startTls));
    }
}
```

需要注意一点，SslHandler必须要添加到ChannelPipeline的第一个位置，可能有一些例外，但是最好这样来做。回想一下之前讲解的ChannelHandler，ChannelPipeline就像是一个在处理“入站”数据时先进先出，在处理“出站”数据时后进先出的队列。最先添加的SslHandler会啊在其他Handler处理逻辑数据之前对数据进行加密，从而确保Netty服务端的所有的Handler的变化都是安全的。

SslHandler提供了一些有用的方法，可以用来修改其行为或得到通知，一旦SSL/TLS完成握手(在握手过程中的两个对等通道互相验证对方，然后选择一个加密密码)，SSL/TLS是自动执行的。看下面方法列表：

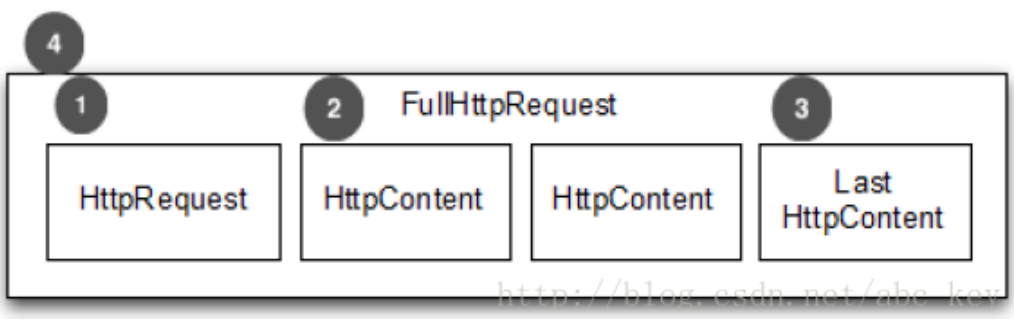
- setHandshakeTimeout(long handshakeTimeout, TimeUnit unit)，设置握手超时时间，ChannelFuture将得到通知
- setHandshakeTimeoutMillis(long handshakeTimeoutMillis)，设置握手超时时间，ChannelFuture将得到通知
- getHandshakeTimeoutMillis()，获取握手超时时间值
- setCloseNotifyTimeout(long closeNotifyTimeout, TimeUnit unit)，设置关闭通知超时时间，若超时，ChannelFuture会关闭失败
- setHandshakeTimeoutMillis(long handshakeTimeoutMillis)，设置关闭通知超时时间，若超时，ChannelFuture会关闭失败
- getCloseNotifyTimeoutMillis()，获取关闭通知超时时间
- handshakeFuture()，返回完成握手后的ChannelFuture
- close()，发送关闭通知请求关闭和销毁

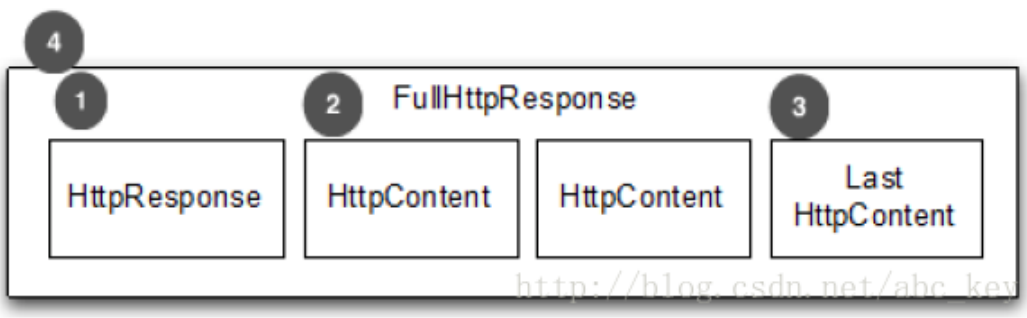
8.2 使用Netty创建HTTP/HTTPS程序

HTTP/HTTPS是最常用的协议之一，可以通过HTTP/HTTPS访问网站，或者是提供对外公开的接口服务等等。Netty附带了使用HTTP/HTTPS的handlers，而不需要我们自己来编写编解码器。

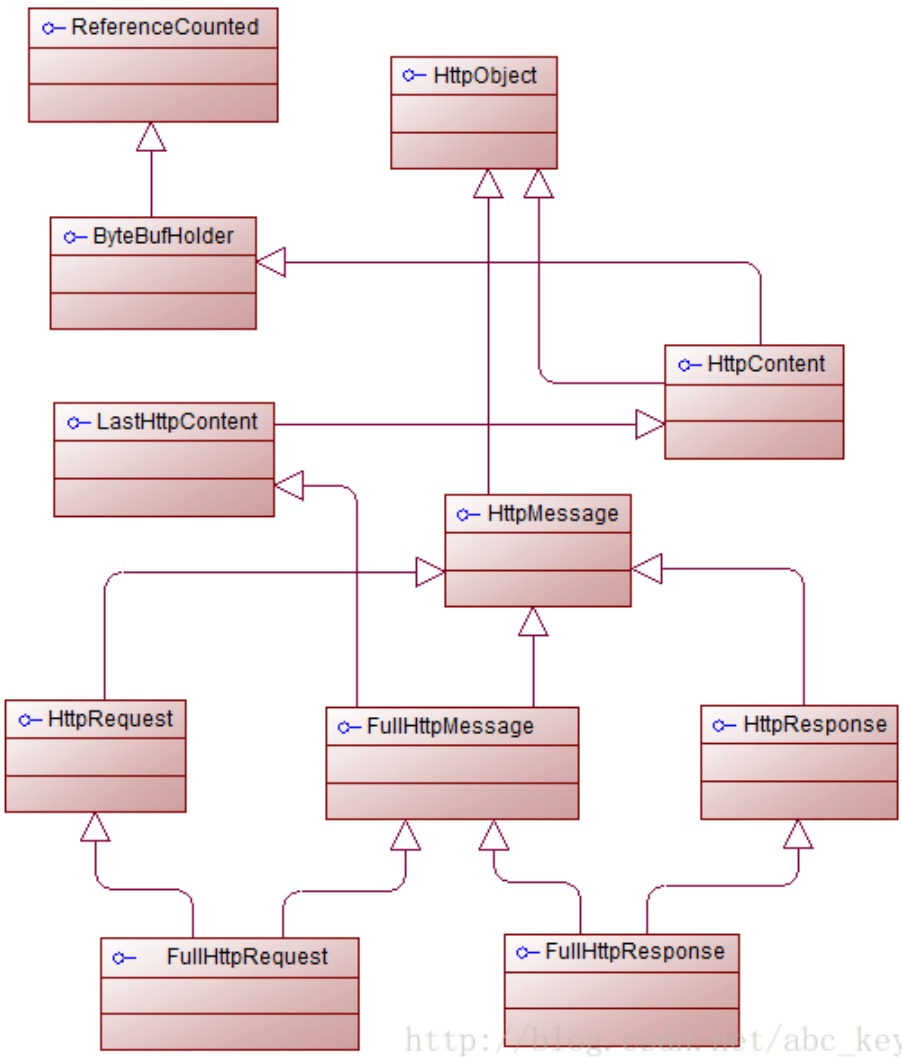
8.2.1 Netty的HTTP编码器，解码器和编解码器

HTTP是请求-响应模式，客户端发送一个http请求，服务就响应此请求。Netty提供了简单的编码解码HTTP协议消息的Handler。下图显示了http请求和响应：





如上面两个图所示，一个HTTP请求/响应消息可能包含不止一个，但最终都会有LastHttpContent消息。FullHttpRequest和FullHttpResponse是Netty提供的两个接口，分别用来完成http请求和响应。所有的HTTP消息类型都实现了HttpObject接口。下面是类关系图：



Netty提供了HTTP请求和响应的编码器和解码器，看下面列表：

- HttpRequestEncoder，将HttpRequest或HttpContent编码成ByteBuf
- HttpRequestDecoder，将ByteBuf解码成HttpRequest和HttpContent
- HttpResponseEncoder，将HttpResponse或HttpContent编码成ByteBuf
- HttpResponseDecoder，将ByteBuf解码成HttpResponse和HttpContent

看下面代码：

```java
public class HttpDecoderEncoderInitializer extends ChannelInitializer<Channel> {

    private final boolean client;

    public HttpDecoderEncoderInitializer(boolean client) {
        this.client = client;
    }

    @Override
    protected void initChannel(Channel ch) throws Exception {
        ChannelPipeline pipeline = ch.pipeline();
        if (client) {
            pipeline.addLast("decoder", new HttpResponseDecoder());
            pipeline.addLast("", new HttpRequestEncoder());
        } else {
            pipeline.addLast("decoder", new HttpRequestDecoder());
            pipeline.addLast("encoder", new HttpResponseEncoder());
        }
    }
}
```

如果你需要在ChannelPipeline中有一个解码器和编码器，还分别有一个在客户端和服务器简单的编解码器：HttpClientCodec和HttpServerCodec。

在ChannelPipelien中有解码器和编码器(或编解码器)后就可以操作不同的HttpObject消息了；但是HTTP请求和响应可以有很多消息数据，你需要处理不同的部分，可能也需要聚合这些消息数据，这是很麻烦的。为了解决这个问题，Netty提供了一个聚合器，它将消息部分合并到FullHttpRequest和FullHttpResponse，因此不需要担心接收碎片消息数据。

8.2.2 HTTP消息聚合

处理HTTP时可能接收HTTP消息片段，Netty需要缓冲直到接收完整个消息。要完成的处理HTTP消息，并且内存开销也不会很大，Netty为此提供了HttpObjectAggregator。通过HttpObjectAggregator，Netty可以聚合HTTP消息，使用FullHttpResponse和FullHttpRequest到

ChannelPipeline中的下一个ChannelHandler，这就消除了断裂消息，保证了消息的完整。下面代码显示了如何聚合：

```java
/**
 * 添加聚合http消息的Handler
 *
 * @author c.k
 *
 */
public class HttpAggregatorInitializer extends ChannelInitializer<Channel> {

    private final boolean client;

    public HttpAggregatorInitializer(boolean client) {
        this.client = client;
    }

    @Override
    protected void initChannel(Channel ch) throws Exception {
        ChannelPipeline pipeline = ch.pipeline();
        if (client) {
            pipeline.addLast("codec", new HttpClientCodec());
        } else {
            pipeline.addLast("codec", new HttpServerCodec());
        }
        pipeline.addLast("aggegator", new HttpObjectAggregator(512 * 1024));
    }

}
```

如上面代码，很容使用Netty自动聚合消息。但是请注意，为了防止Dos攻击服务器，需要合理的限制消息的大小。应设置多大取决于实际的需求，当然也得有足够的内存可用。

8.2.3 HTTP压缩

使用HTTP时建议压缩数据以减少传输流量，压缩数据会增加CPU负载，现在的硬件设施都很强大，大多数时候压缩数据时一个好主意。Netty支持“gzip”和“deflate”，为此提供了两个ChannelHandler实现分别用于压缩和解压。看下面代码：

```java
@Override
protected void initChannel(Channel ch) throws Exception {
    ChannelPipeline pipeline = ch.pipeline();
    if (client) {
        pipeline.addLast("codec", new HttpClientCodec());
        //添加解压缩Handler
        pipeline.addLast("decompressor", new HttpContentDecompressor());
    } else {
        pipeline.addLast("codec", new HttpServerCodec());
        //添加解压缩Handler
        pipeline.addLast("decompressor", new HttpContentDecompressor());
    }
    pipeline.addLast("aggegator", new HttpObjectAggregator(512 * 1024));
}
```

8.2.4 使用HTTPS

网络中传输的重要数据需要加密来保护，使用Netty提供的SslHandler可以很容易实现，看下面代码：

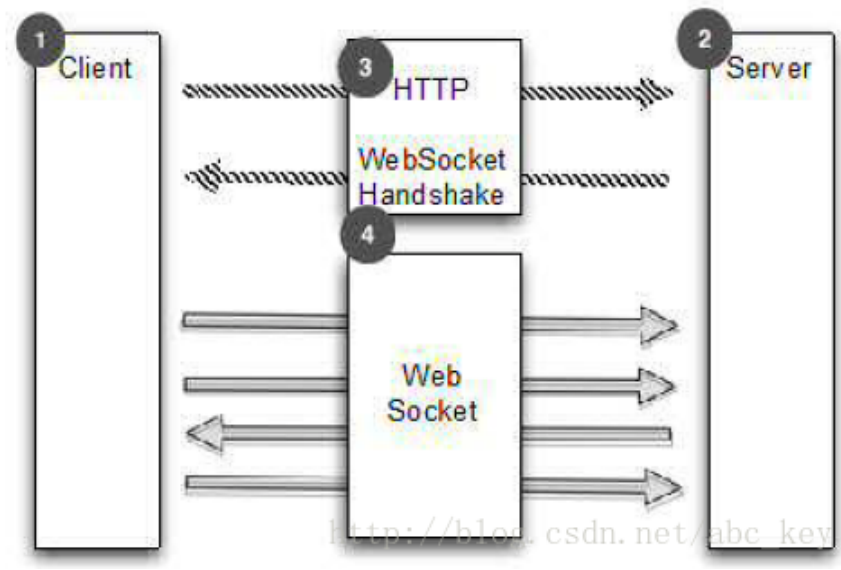
```java
/**
 * 使用SSL对HTTP消息加密
 *
 * @author c.k
 *
 */
public class HttpsCodecInitializer extends ChannelInitializer<Channel> {

    private final SSLContext context;
    private final boolean client;

    public HttpsCodecInitializer(SSLContext context, boolean client) {
        this.context = context;
        this.client = client;
    }

    @Override
```

```
18.     protected void initChannel(Channel ch) throws Exception {
19.         SSLEngine engine = context.createSSLEngine();
20.         engine.setUseClientMode(client);
21.         ChannelPipeline pipeline = ch.pipeline();
22.         pipeline.addFirst("ssl", new SslHandler(engine));
23.         if (client) {
24.             pipeline.addLast("codec", new HttpClientCodec());
25.         } else {
26.             pipeline.addLast("codec", new HttpServerCodec());
27.         }
28.     }
29.
30. }
```

8.2.5 WebSocket

HTTP是不错的协议，但是如果需要实时发布信息怎么做？有个做法就是客户端一直轮询请求服务器，这种方式虽然可以达到目的，但是其缺点很多，也不是优秀的解决方案，为了解决这个问题，便出现了WebSocket。

WebSocket允许数据双向传输，而不需要请求-响应模式。早期的WebSocket只能发送文本数据，然后现在不仅可以发送文本数据，也可以发送二进制数据，这使得可以使用WebSocket构建你想要的程序。下图是WebSocket的通信示例图：



在应用程序中添加WebSocket支持很容易，Netty附带了WebSocket的支持，通过ChannelHandler来实现。使用WebSocket有不同的消息类型需要处理。下面列表列出了Netty中WebSocket类型：

- BinaryWebSocketFrame，包含二进制数据
- TextWebSocketFrame，包含文本数据
- ContinuationWebSocketFrame，包含二进制数据或文本数据，BinaryWebSocketFrame和TextWebSocketFrame的结合体
- CloseWebSocketFrame，WebSocketFrame代表一个关闭请求，包含关闭状态码和短语
- PingWebSocketFrame，WebSocketFrame要求PongWebSocketFrame发送数据
- PongWebSocketFrame，WebSocketFrame要求PingWebSocketFrame响应

为了简化，我们只看看如何使用WebSocket服务器。客户端使用可以看Netty自带的WebSocket例子。

Netty提供了许多方法来使用WebSocket，但最简单常用的方法是使用WebSocketServerProtocolHandler。看下面代码：

```
[java]
01. /**
02.  * WebSocket Server，若想使用SSL加密，将SslHandler加载ChannelPipeline的最前面即可
03.  * @author c.k
04.  *
05.  */
06. public class WebSocketServerInitializer extends ChannelInitializer<Channel> {
07.
08.     @Override
09.     protected void initChannel(Channel ch) throws Exception {
10.         ch.pipeline().addLast(new HttpServerCodec(),
11.                 new HttpObjectAggregator(65536),
12.                 new WebSocketServerProtocolHandler("/websocket"),
13.                 new TextFrameHandler(),
14.                 new BinaryFrameHandler(),
15.                 new ContinuationFrameHandler());
16.     }
17.
18.     public static final class TextFrameHandler extends SimpleChannelInboundHandler<TextWebSocketFrame> {
19.         @Override
20.         protected void channelRead0(ChannelHandlerContext ctx, TextWebSocketFrame msg) throws Exception {
21.             // handler text frame
```

```
        }
    }

    public static final class BinaryFrameHandler extends SimpleChannelInboundHandler<BinaryWebSocketFrame>{
        @Override
        protected void channelRead0(ChannelHandlerContext ctx, BinaryWebSocketFrame msg) throws Exception {
            //handler binary frame
        }
    }

    public static final class ContinuationFrameHandler extends SimpleChannelInboundHandler<ContinuationWebSocketFrame>{
        @Override
        protected void channelRead0(ChannelHandlerContext ctx, ContinuationWebSocketFrame msg) throws Exception {
            //handler continuation frame
        }
    }
}
```

8.2.6 SPDY

SPDY（读作“SPeeDY”）是Google开发的基于TCP的应用层协议，用以最小化网络延迟，提升网络速度，优化用户的网络使用体验。SPDY并不是一种用于替代HTTP的协议，而是对HTTP协议的增强。新协议的功能包括数据流的多路复用、请求优先级以及HTTP报头压缩。谷歌表示，引入SPDY协议后，在实验室测试中页面加载速度比原先快64%。

SPDY的定位：

- 将页面加载时间减少50%。
- 最大限度地减少部署的复杂性。SPDY使用TCP作为传输层，因此无需改变现有的网络设施。
- 避免网站开发者改动内容。 支持SPDY唯一需要变化的是客户端代理和Web服务器应用程序。

SPDY实现技术：

- 单个TCP连接支持并发的HTTP请求。
- 压缩报头和去掉不必要的头部来减少当前HTTP使用的带宽。
- 定义一个容易实现，在服务器端高效率的协议。通过减少边缘情况、定义易解析的消息格式来减少HTTP的复杂性。
- 强制使用SSL，让SSL协议在现存的网络设施下有更好的安全性和兼容性。
- 允许服务器在需要时发起对客户端的连接并推送数据。

SPDY具体的细节知识及使用可以查阅相关资料，这里不作赘述了。

8.3 处理空闲连接和超时

处理空闲连接和超时是网络应用程序的核心部分。当发送一条消息后，可以检测连接是否还处于活跃状态，若很长时间没用了就可以断开连接。Netty提供了很好的解决方案，有三种不同的ChannelHandler处理闲置和超时连接：

- IdleStateHandler，当一个通道没有进行读写或运行了一段时间后出发IdleStateEvent
- ReadTimeoutHandler，在指定时间内没有接收到任何数据将抛出ReadTimeoutException
- WriteTimeoutHandler，在指定时间内有写入数据将抛出WriteTimeoutException

最常用的是IdleStateHandler，下面代码显示了如何使用IdleStateHandler，如果60秒内没有接收数据或发送数据，操作将失败，连接将关闭：

```java
public class IdleStateHandlerInitializer extends ChannelInitializer<Channel> {

    @Override
    protected void initChannel(Channel ch) throws Exception {
        ChannelPipeline pipeline = ch.pipeline();
        pipeline.addLast(new IdleStateHandler(0, 0, 60, TimeUnit.SECONDS));
        pipeline.addLast(new HeartbeatHandler());
    }

    public static final class HeartbeatHandler extends ChannelInboundHandlerAdapter {
        private static final ByteBuf HEARTBEAT_SEQUENCE = Unpooled.unreleasableBuffer(Unpooled.copiedBuffer(
                "HEARTBEAT", CharsetUtil.UTF_8));

        @Override
        public void userEventTriggered(ChannelHandlerContext ctx, Object evt) throws Exception {
            if (evt instanceof IdleStateEvent) {
                ctx.writeAndFlush(HEARTBEAT_SEQUENCE.duplicate()).addListener(ChannelFutureListener.CLOSE_ON_FAILURE);
            } else {
                super.userEventTriggered(ctx, evt);
            }
        }
    }
```

```
}
```

8.4 解码分隔符和基于长度的协议

使用Netty时会遇到需要解码以分隔符和长度为基础的协议，本节讲解Netty如何解码这些协议。

8.4.1 分隔符协议

经常需要处理分隔符协议或创建基于它们的协议，例如SMTP、POP3、IMAP、Telnet等等；Netty附带的handlers可以很容易的提取一些序列分隔：

- DelimiterBasedFrameDecoder，解码器，接收ByteBuf由一个或多个分隔符拆分，如NUL或换行符
- LineBasedFrameDecoder，解码器，接收ByteBuf以分割线结束，如"\n"和"\r\n"

下图显示了使用"\r\n"分隔符的处理：

下面代码显示使用LineBasedFrameDecoder提取"\r\n"分隔帧：

```java
/**
 * 处理换行分隔符消息
 * @author c.k
 *
 */
public class LineBasedHandlerInitializer extends ChannelInitializer<Channel> {

    @Override
    protected void initChannel(Channel ch) throws Exception {
        ch.pipeline().addLast(new LineBasedFrameDecoder(65 * 1204), new FrameHandler());
    }

    public static final class FrameHandler extends SimpleChannelInboundHandler<ByteBuf> {
        @Override
        protected void channelRead0(ChannelHandlerContext ctx, ByteBuf msg) throws Exception {
            // do something with the frame
        }
    }
}
```

如果框架的东西除了换行符还有别的分隔符，可以使用DelimiterBasedFrameDecoder，只需要将分隔符传递到构造方法中。如果想实现自己的以分隔符为基础的协议，这些解码器是有用的。例如，现在有个协议，它只处理命令，这些命令由名称和参数形成，名称和参数由一个空格分隔，实现这个需求的代码如下：

```java
/**
 * 自定义以分隔符为基础的协议
 * @author c.k
 *
 */
public class CmdHandlerInitializer extends ChannelInitializer<Channel> {

    @Override
    protected void initChannel(Channel ch) throws Exception {
        ch.pipeline().addLast(new CmdDecoder(65 * 1024), new CmdHandler());
    }

    public static final class Cmd {
        private final ByteBuf name;
        private final ByteBuf args;

        public Cmd(ByteBuf name, ByteBuf args) {
            this.name = name;
            this.args = args;
        }

        public ByteBuf getName() {
            return name;
        }
```

```
        public ByteBuf getArgs() {
            return args;
        }
    }

    public static final class CmdDecoder extends LineBasedFrameDecoder {

        public CmdDecoder(int maxLength) {
            super(maxLength);
        }

        @Override
        protected Object decode(ChannelHandlerContext ctx, ByteBuf buffer) throws Exception {
            ByteBuf frame = (ByteBuf) super.decode(ctx, buffer);
            if (frame == null) {
                return null;
            }
            int index = frame.indexOf(frame.readerIndex(), frame.writerIndex(), (byte) ' ');
            return new Cmd(frame.slice(frame.readerIndex(), index), frame.slice(index + 1, frame.writerIndex()));
        }
    }

    public static final class CmdHandler extends SimpleChannelInboundHandler<Cmd> {
        @Override
        protected void channelRead0(ChannelHandlerContext ctx, Cmd msg) throws Exception {
            // do something with the command
        }
    }

}
```

8.4.2 长度为基础的协议

一般经常会碰到以长度为基础的协议，对于这种情况Netty有两个不同的解码器可以帮助我们来解码：

- FixedLengthFrameDecoder
- LengthFieldBasedFrameDecoder

下图显示了FixedLengthFrameDecoder的处理流程：

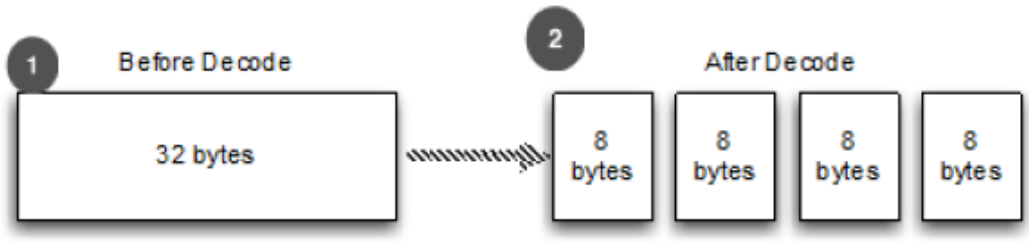


Figure 8.6 Message of fixed size of 8 bytes

如上图所示，FixedLengthFrameDecoder提取固定长度，例子中的是8字节。大部分时候帧的大小被编码在头部，这种情况可以使用LengthFieldBasedFrameDecoder，它会读取头部长度并提取帧的长度。下图显示了它是如何工作的：

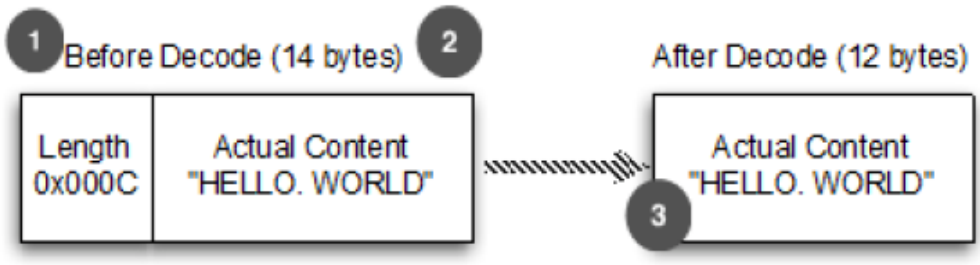


Figure 8.7 Message that has fixed size encoded in the header

如果长度字段是提取框架的一部分，可以在LengthFieldBasedFrameDecoder的构造方法中配置，还可以指定提供的长度。FixedLengthFrameDecoder很容易使用，我们重点讲解LengthFieldBasedFrameDecoder。下面代码显示如何使用LengthFieldBasedFrameDecoder提取8字节长度：

```java
public class LengthBasedInitializer extends ChannelInitializer<Channel> {

    @Override
    protected void initChannel(Channel ch) throws Exception {
        ch.pipeline().addLast(new LengthFieldBasedFrameDecoder(65*1024, 0, 8))
            .addLast(new FrameHandler());
    }

```

```
09.     public static final class FrameHandler extends SimpleChannelInboundHandler<ByteBuf>{
10.         @Override
11.         protected void channelRead0(ChannelHandlerContext ctx, ByteBuf msg) throws Exception {
12.             //do something with the frame
13.         }
14.     }
15. }
```

8.5 写大数据

写大量的数据的一个有效的方法是使用异步框架，如果内存和网络都处于饱满负荷状态，你需要停止写，否则会报OutOfMemoryError。Netty提供了写文件内容时zero-memory-copy机制，这种方法再将文件内容写到网络堆栈空间时可以获得最大的性能。使用零拷贝写文件的内容时通过DefaultFileRegion、ChannelHandlerContext、ChannelPipeline，看下面代码：

```
[java]
01. @Override
02. public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {
03.     File file = new File("test.txt");
04.     FileInputStream fis = new FileInputStream(file);
05.     FileRegion region = new DefaultFileRegion(fis.getChannel(), 0, file.length());
06.     Channel channel = ctx.channel();
07.     channel.writeAndFlush(region).addListener(new ChannelFutureListener() {
08.
09.         @Override
10.         public void operationComplete(ChannelFuture future) throws Exception {
11.             if(!future.isSuccess()){
12.                 Throwable cause = future.cause();
13.                 // do something
14.             }
15.         }
16.     });
17. }
```

如果只想发送文件中指定的数据块应该怎么做呢？Netty提供了ChunkedWriteHandler，允许通过处理ChunkedInput来写大的数据块。下面是ChunkedInput的一些实现类：

- ChunkedFile
- ChunkedNioFile
- ChunkedStream
- ChunkedNioStream

看下面代码：

```
[java]
01. public class ChunkedWriteHandlerInitializer extends ChannelInitializer<Channel> {
02.     private final File file;
03.
04.     public ChunkedWriteHandlerInitializer(File file) {
05.         this.file = file;
06.     }
07.
08.     @Override
09.     protected void initChannel(Channel ch) throws Exception {
10.         ch.pipeline().addLast(new ChunkedWriteHandler())
11.             .addLast(new WriteStreamHandler());
12.     }
13.
14.     public final class WriteStreamHandler extends ChannelInboundHandlerAdapter {
15.         @Override
16.         public void channelActive(ChannelHandlerContext ctx) throws Exception {
17.             super.channelActive(ctx);
18.             ctx.writeAndFlush(new ChunkedStream(new FileInputStream(file)));
19.         }
20.     }
21. }
```

8.6 序列化数据

开发网络程序过程中，很多时候需要传输结构化对象数据POJO,Java中提供了ObjectInputStream和ObjectOutputStream及其他的一些对象序列化接口。Netty中提供基于JDK序列化接口的序列化接口。

8.6.1 普通的JDK序列化

如果你使用ObjectInputStream和ObjectOutputStream，并且需要保持兼容性，不想有外部依赖，那么JDK的序列化是首选。Netty提供了

下面的一些接口，这些接口放在io.netty.handler.codec.serialization包下面：

- CompatibleObjectEncoder
- CompactObjectInputStream
- CompactObjectOutputStream
- ObjectEncoder
- ObjectDecoder
- ObjectEncoderOutputStream
- ObjectDecoderInputStream

8.6.2 通过JBoss编组序列化

如果你想使用外部依赖的接口，JBoss编组是个好方法。JBoss Marshalling序列化的速度是JDK的3倍，并且序列化的结构更紧凑，从而使序列化后的数据更小。Netty附带了JBoss编组序列化的实现，这些实现接口放在io.netty.handler.codec.marshalling包下面：

- CompatibleMarshallingEncoder
- CompatibleMarshallingDecoder
- MarshallingEncoder
- MarshallingDecoder

看下面代码：

```java
/**
 * 使用JBoss Marshalling
 * @author c.k
 *
 */
public class MarshallingInitializer extends ChannelInitializer<Channel> {
    private final MarshallerProvider marshallerProvider;
    private final UnmarshallerProvider unmarshallerProvider;

    public MarshallingInitializer(MarshallerProvider marshallerProvider, UnmarshallerProvider unmarshallerProvider) {
        this.marshallerProvider = marshallerProvider;
        this.unmarshallerProvider = unmarshallerProvider;
    }

    @Override
    protected void initChannel(Channel ch) throws Exception {
        ch.pipeline().addLast(new MarshallingDecoder(unmarshallerProvider))
            .addLast(new MarshallingEncoder(marshallerProvider))
            .addLast(new ObjectHandler());
    }

    public final class ObjectHandler extends SimpleChannelInboundHandler<Serializable> {
        @Override
        protected void channelRead0(ChannelHandlerContext ctx, Serializable msg) throws Exception {
            // do something
        }
    }
}
```

8.6.3 使用ProtoBuf序列化

最有一个序列化方案是Netty附带的ProtoBuf。protobuf是Google开源的一种编码和解码技术，它的作用是使序列化数据更高效。并且谷歌提供了protobuf的不同语言的实现，所以protobuf在跨平台项目中是非常好的选择。Netty附带的protobuf放在io.netty.handler.codec.protobuf包下面：

- ProtobufDecoder
- ProtobufEncoder
- ProtobufVarint32FrameDecoder
- ProtobufVarint32LengthFieldPrepender

看下面代码：

```java
/**
 * 使用protobuf序列化数据，进行编码解码
 * 注意：使用protobuf需要protobuf-java-2.5.0.jar
```

```
 * @author Administrator
 *
 */
public class ProtoBufInitializer extends ChannelInitializer<Channel> {

    private final MessageLite lite;

    public ProtoBufInitializer(MessageLite lite) {
        this.lite = lite;
    }

    @Override
    protected void initChannel(Channel ch) throws Exception {
        ch.pipeline().addLast(new ProtobufVarint32FrameDecoder())
            .addLast(new ProtobufEncoder())
            .addLast(new ProtobufDecoder(lite))
            .addLast(new ObjectHandler());
    }

    public final class ObjectHandler extends SimpleChannelInboundHandler<Serializable> {
        @Override
        protected void channelRead0(ChannelHandlerContext ctx, Serializable msg) throws Exception {
            // do something
        }
    }
}
```