# 【第11章-预置的ChannelHandler和解码器】

Netty 为许多通用协议提供了编解码器和处理器，几乎可以开箱即用，这减少了你在那些相当繁琐的事务上本来会花费的时间与精力。在本章中，我们将探讨这些工具以及它们所带来的好处，其中包括Netty 对于SSL/TLS 和WebSocket 的支持，以及如何简单地通过数据压缩来压榨HTTP，以获取更好的性能。

## 11.1 通过SSL/TLS 保护Netty 应用程序

如今，数据隐私是一个非常值得关注的问题，作为开发人员，我们需要准备好应对它。至少，我们应该熟悉像SSL和TLS这样的安全协议，它们层叠在其他协议之上，用以实现数据安全。我们在访问安全网站时遇到过这些协议，但是它们也可用于其他不是基于HTTP的应用程序，如安全SMTP（SMTPS）邮件服务器甚至是关系型数据库系统。

为了支持SSL/TLS，Java 提供了javax.net.ssl 包，它的SSLContext 和SSLEngine类使得实现解密和加密相当简单直接。Netty 通过一个名为SslHandler 的ChannelHandler实现利用了这个API，其中SslHandler 在内部使用SSLEngine 来完成实际的工作。

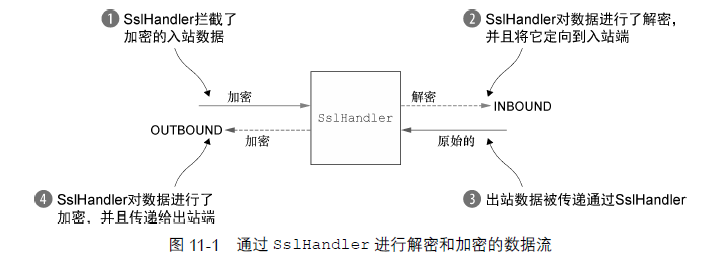
**Netty 的OpenSSL/SSLEngine 实现**

Netty 还提供了使用OpenSSL 工具包（www.openssl.org）的SSLEngine 实现。这个OpenSslEngine 类提供了比JDK 提供的SSLEngine 实现更好的性能。

如果OpenSSL库可用，可以将Netty 应用程序（客户端和服务器）配置为默认使用OpenSslEngine。如果不可用，Netty 将会回退到JDK 实现。有关配置OpenSSL 支持的详细说明，参见Netty 文档：http://netty.io/wiki/forked-tomcat-native.html#wikih2-1。

注意，无论你使用JDK 的SSLEngine 还是使用Netty 的OpenSslEngine，SSL API 和数据流都是一致的。

图11-1 展示了使用SslHandler 的数据流。



代码清单11-1 展示了如何使用ChannelInitializer 来将SslHandler添加到ChannelPipeline 中。回想一下，ChannelInitializer 用于在Channel 注册好时设置ChannelPipeline。

|  |
| --- |
| **// 代码清单11-1 添加SSL/TLS 支持**  **public class** SslChannelInitializer **extends** ChannelInitializer<Channel> {  **private final** SslContext **context**;  **private final boolean startTls**;   *// 传入要使用的SslContext  // 如果设置为true，第一个写入的消息将不会被加密（客户端应该设置为true）* **public** SslChannelInitializer(SslContext context, **boolean** startTls) {  **this**.**context** = context;  **this**.**startTls** = startTls;  }   @Override  **protected void** initChannel(Channel ch) **throws** Exception {  *// 对于每个SslHandler 实例，都使用Channel 的ByteBufAllocator  // 从SslContext获取一个新的SSLEngine* SSLEngine engine = **context**.newEngine(ch.alloc());  *// 将SslHandler 作为第一个ChannelHandler 添加到ChannelPipeline 中* ch.pipeline().addFirst(**"ssl"**, **new** SslHandler(engine, **startTls**));  } } |

在大多数情况下，SslHandler 将是ChannelPipeline 中的第一个ChannelHandler。这确保了只有在所有其他的ChannelHandler 将它们的逻辑应用到数据之后，才会进行加密。

SslHandler 具有一些有用的方法，如表11-1 所示。例如，在握手阶段，两个节点将相互验证并且商定一种加密方式。你可以通过配置SslHandler 来修改它的行为，或者在SSL/TLS握手一旦完成之后提供通知，握手阶段完成之后，所有的数据都将会被加密。SSL/TLS 握手将会被自动执行。



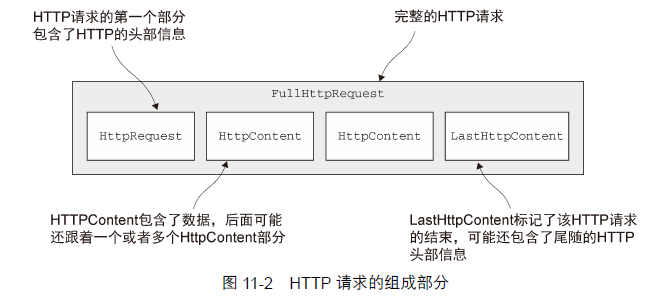
## 11.2 构建基于Netty 的HTTP/HTTPS 应用程序

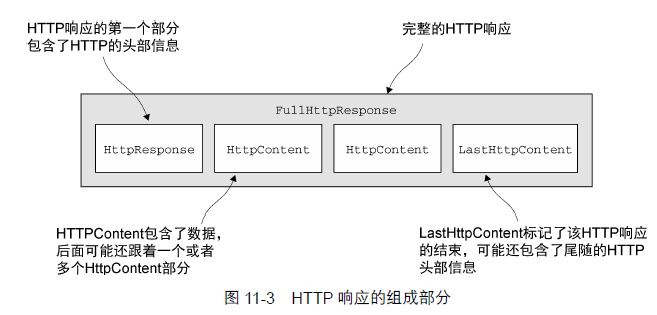
HTTP/HTTPS 是最常见的协议套件之一，并且随着智能手机的成功，它的应用也日益广泛，因为对于任何公司来说，拥有一个可以被移动设备访问的网站几乎是必须的。这些协议也被用于其他方面。许多组织导出的用于和他们的商业合作伙伴通信的WebService API 一般也是基于HTTP（S）的。

接下来，我们来看看Netty 提供的ChannelHandler，你可以用它来处理HTTP 和HTTPS协议，而不必编写自定义的编解码器。

### 11.2.1 HTTP 解码器、编码器和编解码器

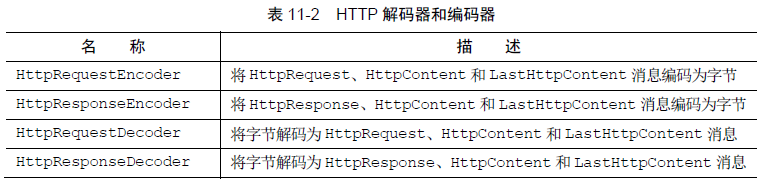
HTTP 是基于请求/响应模式的：客户端向服务器发送一个HTTP 请求，然后服务器将会返回一个HTTP 响应。Netty 提供了多种编码器和解码器以简化对这个协议的使用。图11-2 和图11-3分别展示了生产和消费HTTP 请求和HTTP 响应的方法。





如图11-2 和图11-3 所示，一个HTTP 请求/响应可能由多个数据部分组成，并且它总是以一个LastHttpContent 部分作为结束。FullHttpRequest 和FullHttpResponse 消息是特殊的子类型，分别代表了完整的请求和响应。所有类型的HTTP 消息（FullHttpRequest、LastHttpContent 以及代码清单11-2 中展示的那些）都实现了HttpObject 接口。

表11-2 概要地介绍了处理和生成这些消息的HTTP 解码器和编码器。



代码清单11-2 中的HttpPipelineInitializer 类展示了将HTTP 支持添加到你的应用程序是多么简单—几乎只需要将正确的ChannelHandler 添加到ChannelPipeline 中。

|  |
| --- |
| **// 代码清单11-2 添加HTTP 支持**  **public class** HttpPipelineInitializer **extends** ChannelInitializer<Channel> {  **private final boolean client**;   **public** HttpPipelineInitializer(**boolean** client) {  **this**.**client** = client;  }   @Override  **protected void** initChannel(Channel ch) **throws** Exception {  ChannelPipeline pipeline = ch.pipeline();  **if** (**client**) {  *// 如果是客户端，则添加HttpResponseDecoder以处理来自服务器的响应  // 如果是客户端，则添加HttpRequestEncoder以向服务器发送请求* pipeline.addLast(**"decoder"**, **new** HttpResponseDecoder());  pipeline.addLast(**"encoder"**, **new** HttpRequestEncoder());  } **else** {  *// 如果是服务器，则添加HttpResponseEncoder以向客户端发送响应  // 如果是服务器，则添加HttpRequestDecoder以接收来自客户端的请求* pipeline.addLast(**"decoder"**, **new** HttpRequestDecoder());  pipeline.addLast(**"encoder"**, **new** HttpResponseEncoder());  }  } } |

### 11.2.2 聚合HTTP 消息

在ChannelInitializer 将ChannelHandler 安装到ChannelPipeline 中之后，你便可以处理不同类型的HttpObject 消息了。但是由于HTTP 的请求和响应可能由许多部分组成，因此你需要聚合它们以形成完整的消息。为了消除这项繁琐的任务，Netty 提供了一个聚合器，它可以将多个消息部分合并为FullHttpRequest 或者FullHttpResponse 消息。通过这样的方式，你将总是看到完整的消息内容。

由于消息分段需要被缓冲，直到可以转发一个完整的消息给下一个ChannelInboundHandler，所以这个操作有轻微的开销。其所带来的好处便是你不必关心消息碎片了。

引入这种自动聚合机制只不过是向ChannelPipeline 中添加另外一个ChannelHandler罢了。代码清单11-3 展示了如何做到这一点。

|  |
| --- |
| **// 代码清单11-3 自动聚合HTTP 的消息片段**  **public class** HttpAggregatorInitializer **extends** ChannelInitializer<Channel> {  **private final boolean isClient**;   **public** HttpAggregatorInitializer(**boolean** isClient) {  **this**.**isClient** = isClient;  }   @Override  **protected void** initChannel(Channel ch) **throws** Exception {  ChannelPipeline pipeline = ch.pipeline();  **if** (**isClient**) {  *// 如果是客户端，则添加HttpClientCodec* pipeline.addLast(**"codec"**, **new** HttpClientCodec());  } **else** {  *// 如果是服务器，则添加HttpServerCodec* pipeline.addLast(**"codec"**, **new** HttpServerCodec());  }    *// 将最大的消息大小为512 KB的HttpObjectAggregator 添加到ChannelPipeline* pipeline.addLast(**"aggregator"**, **new** HttpObjectAggregator(**512** \* **1024**));  } } |

### 11.2.3 HTTP 压缩

当使用HTTP 时，建议开启压缩功能以尽可能多地减小传输数据的大小。虽然压缩会带来一些CPU 时钟周期上的开销，但是通常来说它都是一个好主意，特别是对于文本数据来说。

Netty 为压缩和解压缩提供了ChannelHandler 实现，它们同时支持gzip 和deflate 编码。

**HTTP 请求的头部信息**

客户端可以通过提供以下头部信息来指示服务器它所支持的压缩格式：

GET /encrypted-area HTTP/1.1

Host: www.example.com

Accept -Encoding: gzip, deflate

然而，需要注意的是，服务器没有义务压缩它所发送的数据。

代码清单11-4 展示了一个例子。

|  |
| --- |
| **// 代码清单11-4 自动压缩HTTP 消息**  **public class** HttpCompressionInitializer **extends** ChannelInitializer<Channel> {  **private final boolean isClient**;   **public** HttpCompressionInitializer(**boolean** isClient) {  **this**.**isClient** = isClient;  }   @Override  **protected void** initChannel(Channel ch) **throws** Exception {  ChannelPipeline pipeline = ch.pipeline();  **if** (**isClient**) {  *// 如果是客户端，则添加HttpClientCodec  // 如果是客户端，则添加HttpContentDecompressor 以处理来自服务器的压缩内容* pipeline.addLast(**"codec"**, **new** HttpClientCodec());  pipeline.addLast(**"decompressor"**, **new** HttpContentDecompressor());  } **else** {  *// 如果是服务器，则添加HttpServerCodec  // 如果是服务器，则添加HttpContentCompressor来压缩数据（如果客户端支持它）* pipeline.addLast(**"codec"**, **new** HttpServerCodec());  pipeline.addLast(**"compressor"**, **new** HttpContentCompressor());  }  } } |

**压缩及其依赖**

如果你正在使用的是JDK 6 或者更早的版本，那么你需要将JZlib（www.jcraft.com/jzlib/）添加到CLASSPATH 中以支持压缩功能。对于Maven，请添加以下依赖项：

<dependency>

<groupId>com.jcraft</groupId>

<artifactId>jzlib</artifactId>

<version>1.1.3</version>

</depe ndency>

### 11.2.4 使用HTTPS

代码清单11-5 显示，启用HTTPS 只需要将SslHandler 添加到ChannelPipeline 的ChannelHandler 组合中。

|  |
| --- |
| **// 代码清单11-5 使用HTTPS**  **public class** HttpsCodecInitializer **extends** ChannelInitializer<Channel> {  **private final** SslContext **context**;  **private final boolean isClient**;   **public** HttpsCodecInitializer(SslContext context, **boolean** isClient) {  **this**.**context** = context;  **this**.**isClient** = isClient;  }   @Override  **protected void** initChannel(Channel ch) **throws** Exception {  ChannelPipeline pipeline = ch.pipeline();  SSLEngine engine = **context**.newEngine(ch.alloc());  *// 将SslHandler 添加到ChannelPipeline 中以使用HTTPS* pipeline.addFirst(**"ssl"**, **new** SslHandler(engine));   **if** (**isClient**) {  *// 如果是客户端，则添加HttpClientCodec* pipeline.addLast(**"codec"**, **new** HttpClientCodec());  } **else** {  *// 如果是服务器，则添加HttpServerCodec* pipeline.addLast(**"codec"**, **new** HttpServerCodec());  }  } } |

前面的代码是一个很好的例子，说明了Netty 的架构方式是如何将代码重用变为杠杆作用的。只需要简单地将一个ChannelHandler 添加到ChannelPipeline 中，便可以提供一项新功能，甚至像加密这样重要的功能都能提供。

### 11.2.5 WebSocket

Netty 针对基于HTTP 的应用程序的广泛工具包中包括了对它的一些最先进的特性的支持。在这一节中，我们将探讨WebSocket ——一种在2011 年被互联网工程任务组（IETF）标准化的协议。

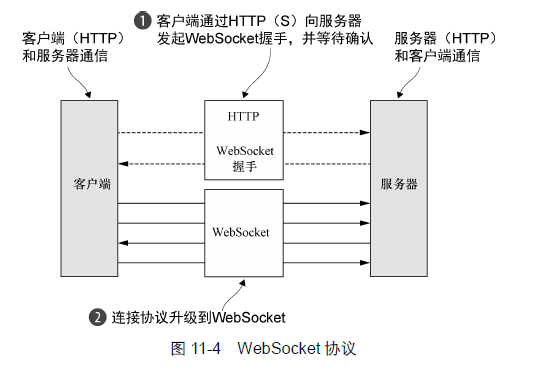
WebSocket解决了一个长期存在的问题：既然底层的协议（HTTP）是一个请求/响应模式的交互序列，那么如何实时地发布信息呢？AJAX提供了一定程度上的改善，但是数据流仍然是由客户端所发送的请求驱动的。还有其他的一些或多或少的取巧方式，但是最终它们仍然属于扩展性受限的变通之法。

WebSocket规范以及它的实现代表了对一种更加有效的解决方案的尝试。简单地说，WebSocket提供了“在一个单个的TCP连接上提供双向的通信……结合WebSocket API……它为网页和远程服务器之间的双向通信提供了一种替代HTTP轮询的方案。”

也就是说，WebSocket 在客户端和服务器之间提供了真正的双向数据交换。我们不会深入地描述太多的内部细节，但是我们还是应该提到，尽管最早的实现仅限于文本数据，但是现在已经不是问题了；WebSocket 现在可以用于传输任意类型的数据，很像普通的套接字。

图11-4 给出了WebSocket 协议的一般概念。在这个场景下，通信将作为普通的HTTP 协议开始，随后升级到双向的WebSocket 协议。

要想向你的应用程序中添加对于WebSocket 的支持，你需要将适当的客户端或者服务器WebSocket ChannelHandler 添加到ChannelPipeline 中。这个类将处理由WebSocket 定义的称为帧的特殊消息类型。如表11-3 所示，WebSocketFrame 可以被归类为数据帧或者控制帧。





因为Netty主要是一种服务器端的技术，所以在这里我们重点创建WebSocket服务器，。代码清单11-6 展示了一个使用WebSocketServerProtocolHandler的简单示例，这个类处理协议升级握手，以及3 种控制帧——Close、Ping和Pong。Text和Binary数据帧将会被传递给下一个（由你实现的）ChannelHandler进行处理。

|  |
| --- |
| **// 代码清单11-6 在服务器端支持WebSocket**  **public class** WebSocketServerInitializer **extends** ChannelInitializer<Channel> {  @Override  **protected void** initChannel(Channel ch) **throws** Exception {  ch.pipeline().addLast(  **new** HttpServerCodec(),  *// 为握手提供聚合的HttpRequest* **new** HttpObjectAggregator(**65536**),  *// 如果被请求的端点是"/websocket"，则处理该升级握手* **new** WebSocketServerProtocolHandler(**"/websocket"**),  *// TextFrameHandler 处理TextWebSocketFrame* **new** TextFrameHandler(),  *// BinaryFrameHandler 处理BinaryWebSocketFrame* **new** BinaryFrameHandler(),  *// ContinuationFrameHandler 处理ContinuationWebSocketFrame* **new** ContinuationFrameHandler());  }   **public static final class** TextFrameHandler **extends** SimpleChannelInboundHandler<TextWebSocketFrame> {  @Override  **public void** channelRead0(ChannelHandlerContext ctx,  TextWebSocketFrame msg) **throws** Exception {  *// Handle text frame* }  }   **public static final class** BinaryFrameHandler **extends** SimpleChannelInboundHandler<BinaryWebSocketFrame> {  @Override  **public void** channelRead0(ChannelHandlerContext ctx,  BinaryWebSocketFrame msg) **throws** Exception {  *// Handle binary frame* }  }   **public static final class** ContinuationFrameHandler **extends** SimpleChannelInboundHandler<ContinuationWebSocketFrame> {  @Override  **public void** channelRead0(ChannelHandlerContext ctx,  ContinuationWebSocketFrame msg) **throws** Exception {  *// Handle continuation frame* }  } } |

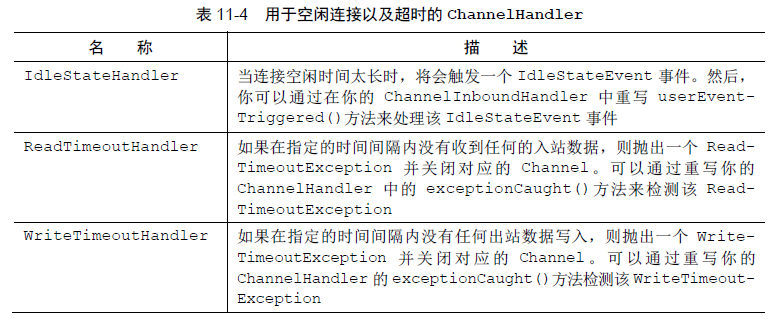
**保护WebSocket**

要想为WebSocket 添加安全性，只需要将SslHandler 作为第一个ChannelHandler 添加到ChannelPipeline 中。

### 11.3 空闲的连接和超时

到目前为止，我们的讨论都集中在Netty 通过专门的编解码器和处理器对HTTP 的变型HTTPS 和WebSocket 的支持上。只要你有效地管理你的网络资源，这些技术就可以使得你的应用程序更加高效、易用和安全。所以，让我们一起来探讨下首先需要关注的——连接管理吧。

检测空闲连接以及超时对于及时释放资源来说是至关重要的。由于这是一项常见的任务，Netty 特地为它提供了几个ChannelHandler 实现。表11-4 给出了它们的概述。



让我们仔细看看在实践中使用得最多的IdleStateHandler 吧。代码清单11-7 展示了当使用通常的发送心跳消息到远程节点的方法时，如果在60 秒之内没有接收或者发送任何的数据，我们将如何得到通知；如果没有响应，则连接会被关闭。

|  |
| --- |
| **// 代码清单11-7 发送心跳**  **public class** IdleStateHandlerInitializer **extends** ChannelInitializer<Channel> {  @Override  **protected void** initChannel(Channel ch) **throws** Exception {  ChannelPipeline pipeline = ch.pipeline();  pipeline.addLast(  *// IdleStateHandler 将在被触发时发送一个IdleStateEvent 事件* **new** IdleStateHandler(**0**, **0**, **60**, TimeUnit.***SECONDS***));  *// 将一个HeartbeatHandler添加到ChannelPipeline中* pipeline.addLast(**new** HeartbeatHandler());  }   *// 实现userEventTriggered()方法以发送心跳消息* **public static final class** HeartbeatHandler  **extends** ChannelInboundHandlerAdapter {  *// 发送到远程节点的心跳消息* **private static final** ByteBuf ***HEARTBEAT\_SEQUENCE*** =  Unpooled.*unreleasableBuffer*(Unpooled.*copiedBuffer*(**"HEARTBEAT"**, CharsetUtil.***ISO\_8859\_1***));   @Override  **public void** userEventTriggered(ChannelHandlerContext ctx,  Object evt) **throws** Exception {  *// 发送心跳消息，并在发送失败时关闭该连接* **if** (evt **instanceof** IdleStateEvent) {  ctx.writeAndFlush(***HEARTBEAT\_SEQUENCE***.duplicate())  .addListener(ChannelFutureListener.***CLOSE\_ON\_FAILURE***);  } **else** {  *// 不是IdleStateEvent事件，所以将它传递给下一个ChannelInboundHandler* **super**.userEventTriggered(ctx, evt);  }  }  } } |

这个示例演示了如何使用IdleStateHandler 来测试远程节点是否仍然还活着，并且在它失活时通过关闭连接来释放资源。

如果连接超过60 秒没有接收或者发送任何的数据，那么IdleStateHandler 将会使用一个IdleStateEvent 事件来调用fireUserEventTriggered()方法。HeartbeatHandler 实现了userEventTriggered()方法，如果这个方法检测到IdleStateEvent 事件，它将会发送心跳消息，并且添加一个将在发送操作失败时关闭该连接的ChannelFutureListener。

## 11.4 解码基于分隔符的协议和基于长度的协议

在使用Netty 的过程中，你将会遇到需要解码器的基于分隔符和帧长度的协议。下一节将解释Netty 所提供的用于处理这些场景的实现。

### 11.4.1 基于分隔符的协议

基于分隔符的（delimited）消息协议使用定义的字符来标记的消息或者消息段（通常被称为帧）的开头或者结尾。由RFC文档正式定义的许多协议（如SMTP、POP3、IMAP以及Telnet）都是这样的。此外，当然，私有组织通常也拥有他们自己的专有格式。无论你使用什么样的协议，表11-5 中列出的解码器都能帮助你定义可以提取由任意标记（token）序列分隔的帧的自定义解码器。

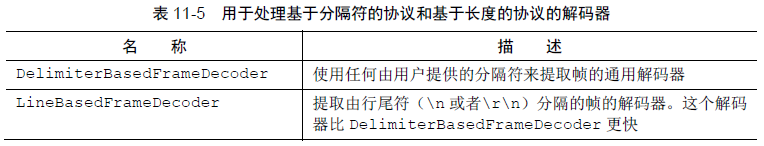
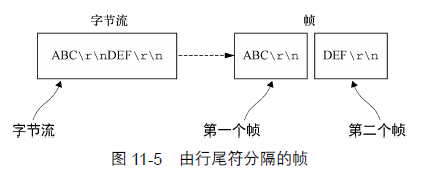


图11-5 展示了当帧由行尾序列\r\n（回车符+换行符）分隔时是如何被处理的。



代码清单11-8 展示了如何使用LineBasedFrameDecoder 来处理图11-5 所示的场景。

|  |
| --- |
| **// 代码清单11-8 处理由行尾符分隔的帧**  **public class** LineBasedHandlerInitializer **extends** ChannelInitializer<Channel> {  @Override  **protected void** initChannel(Channel ch) **throws** Exception {  ChannelPipeline pipeline = ch.pipeline();  *// 该LineBasedFrameDecoder 将提取的帧转发给下一个ChannelInboundHandler* pipeline.addLast(**new** LineBasedFrameDecoder(**64** \* **1024**));  *// 添加FrameHandler以接收帧* pipeline.addLast(**new** FrameHandler());  }   **public static final class** FrameHandler **extends** SimpleChannelInboundHandler<ByteBuf> {  *// 传入了单个帧的内容* @Override  **public void** channelRead0(ChannelHandlerContext ctx, ByteBuf msg) **throws** Exception {  *// Do something with the data extracted from the frame* }  } } |

如果你正在使用除了行尾符之外的分隔符分隔的帧，那么你可以以类似的方式使用DelimiterBasedFrameDecoder，只需要将特定的分隔符序列指定到其构造函数即可。

这些解码器是实现你自己的基于分隔符的协议的工具。作为示例，我们将使用下面的协议规范：

* 传入数据流是一系列的帧，每个帧都由换行符（\n）分隔；
* 每个帧都由一系列的元素组成，每个元素都由单个空格字符分隔；
* 一个帧的内容代表一个命令，定义为一个命令名称后跟着数目可变的参数。

我们用于这个协议的自定义解码器将定义以下类：

* Cmd—将帧（命令）的内容存储在ByteBuf 中，一个ByteBuf 用于名称，另一个用于参数；
* CmdDecoder—从被重写了的decode()方法中获取一行字符串，并从它的内容构建一个Cmd 的实例；
* CmdHandler —从CmdDecoder 获取解码的Cmd 对象，并对它进行一些处理；
* CmdHandlerInitializer —为了简便起见，我们将会把前面的这些类定义为专门的ChannelInitializer 的嵌套类，其将会把这些ChannelInboundHandler 安装到ChannelPipeline 中。

正如将在代码清单11-9 中所能看到的那样，这个解码器的关键是扩展LineBasedFrameDecoder。

|  |
| --- |
| **// 代码清单11-9 使用ChannelInitializer 安装解码器**  **public class** CmdHandlerInitializer **extends** ChannelInitializer<Channel> {  **private static final byte *SPACE*** = (**byte**) **' '**;   @Override  **protected void** initChannel(Channel ch) **throws** Exception {  ChannelPipeline pipeline = ch.pipeline();  *// 添加CmdDecoder 以提取Cmd 对象，并将它转发给下一个ChannelInboundHandler* pipeline.addLast(**new** CmdDecoder(**64** \* **1024**));  *// 添加CmdHandler 以接收和处理Cmd 对象* pipeline.addLast(**new** CmdHandler());  }   *// Cmd POJO* **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 name() {  **return name**;  }   **public** ByteBuf args() {  **return args**;  }  }   **public static final class** CmdDecoder **extends** LineBasedFrameDecoder {  **public** CmdDecoder(**int** maxLength) {  **super**(maxLength);  }   *// 从ByteBuf 中提取由行尾符序列分隔的帧* @Override  **protected** Object decode(ChannelHandlerContext ctx, ByteBuf buffer) **throws** Exception {  *// 从ByteBuf 中提取由行尾符序列分隔的帧* ByteBuf frame = (ByteBuf) **super**.decode(ctx, buffer);  *// 如果输入中没有帧，则返回null* **if** (frame == **null**) {  **return null**;  }   *// 查找第一个空格字符的索引。前面是命令名称，接着是参数* **int** index = frame.indexOf(frame.readerIndex(), frame.writerIndex(), ***SPACE***);  *// 使用包含有命令名称和参数的切片创建新的Cmd 对象* **return new** Cmd(frame.slice(frame.readerIndex(), index),  frame.slice(index + **1**, frame.writerIndex()));  }  }   **public static final class** CmdHandler **extends** SimpleChannelInboundHandler<Cmd> {  *// 处理传经ChannelPipeline的Cmd 对象* @Override  **public void** channelRead0(ChannelHandlerContext ctx, Cmd msg)  **throws** Exception {  *// Do something with the command* }  } } |

### 11.4.2 基于长度的协议

基于长度的协议通过将它的长度编码到帧的头部来定义帧，而不是使用特殊的分隔符来标记它的结束。

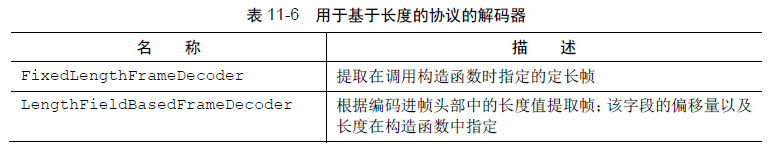
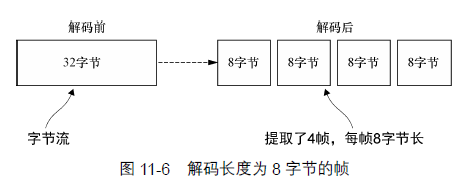
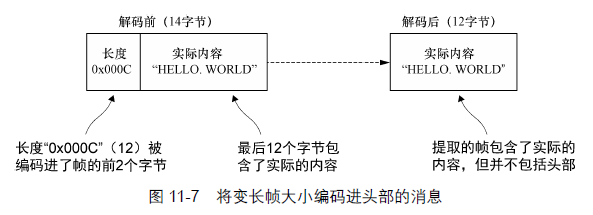


图11-6 展示了FixedLengthFrameDecoder 的功能，其在构造时已经指定了帧长度为8字节。



你将经常会遇到被编码到消息头部的帧大小不是固定值的协议。为了处理这种变长帧，你可以使用LengthFieldBasedFrameDecoder，它将从头部字段确定帧长，然后从数据流中提取指定的字节数。

图11-7 展示了一个示例，其中长度字段在帧中的偏移量为0，并且长度为2 字节。



LengthFieldBasedFrameDecoder 提供了几个构造函数来支持各种各样的头部配置情况。代码清单11-10 展示了如何使用其3 个构造参数分别为maxFrameLength、lengthFieldOffset 和lengthFieldLength 的构造函数。在这个场景中，帧的长度被编码到了帧起始的前8 个字节中。

|  |
| --- |
| **// 代码清单11-10 使用LengthFieldBasedFrameDecoder 解码器基于长度的协议**  **public class** LengthBasedInitializer **extends** ChannelInitializer<Channel> {  @Override  **protected void** initChannel(Channel ch) **throws** Exception {  ChannelPipeline pipeline = ch.pipeline();  pipeline.addLast(  *// 使用LengthFieldBasedFrameDecoder 解码将帧长度编码到帧起始的前8 个字节中的消息* **new** LengthFieldBasedFrameDecoder(**64** \* **1024**, **0**, **8**));  *// 添加FrameHandler以处理每个帧* pipeline.addLast(**new** FrameHandler());  }   **public static final class** FrameHandler **extends** SimpleChannelInboundHandler<ByteBuf> {  *// 处理帧的数据* @Override  **public void** channelRead0(ChannelHandlerContext ctx, ByteBuf msg) **throws** Exception {  *// Do something with the frame* }  } } |

你现在已经看到了Netty 提供的，用于支持那些通过指定协议帧的分隔符或者长度（固定的或者可变的）以定义字节流的结构的协议的编解码器。你将会发现这些编解码器的许多用途，因为许多的常见协议都落到了这些分类之一中。

## 11.5 写大型数据

因为网络饱和的可能性，如何在异步框架中高效地写大块的数据是一个特殊的问题。由于写操作是非阻塞的，所以即使没有写出所有的数据，写操作也会在完成时返回并通知ChannelFuture。当这种情况发生时，如果仍然不停地写入，就有内存耗尽的风险。所以在写大型数据时，需要准备好处理到远程节点的连接是慢速连接的情况，这种情况会导致内存释放的延迟。让我们考虑下将一个文件内容写出到网络的情况。

在我们讨论传输（见4.2 节）的过程中，提到了NIO 的零拷贝特性，这种特性消除了将文件的内容从文件系统移动到网络栈的复制过程。所有的这一切都发生在Netty 的核心中，所以应用程序所有需要做的就是使用一个FileRegion 接口的实现，其在Netty 的API 文档中的定义是：“通过支持零拷贝的文件传输的Channel 来发送的文件区域。”

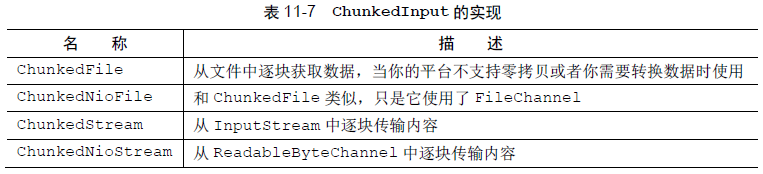
代码清单11-11 展示了如何通过从FileInputStream创建一个DefaultFileRegion，并将其写入Channel（我们甚至可以利用io.netty.channel.ChannelProgressivePromise 来实时获取传输的进度），从而利用零拷贝特性来传输一个文件的内容。

|  |
| --- |
| **// 代码清单11-11 使用FileRegion 传输文件的内容**  **public class** FileRegionWriteHandler **extends** ChannelInboundHandlerAdapter {  **private static final** Channel ***CHANNEL\_FROM\_SOMEWHERE*** = **new** NioSocketChannel();  **private static final** File ***FILE\_FROM\_SOMEWHERE*** = **new** File(**""**);   @Override  **public void** channelActive(**final** ChannelHandlerContext ctx) **throws** Exception {  File file = ***FILE\_FROM\_SOMEWHERE***; *//get reference from somewhere* Channel channel = ***CHANNEL\_FROM\_SOMEWHERE***; *//get reference from somewhere  //...  // 创建一个FileInputStream* FileInputStream in = **new** FileInputStream(file);  *// 以该文件的完整长度创建一个新的DefaultFileRegion* FileRegion region = **new** DefaultFileRegion(in.getChannel(), **0**, file.length());  *// 发送该DefaultFileRegion，并注册一个ChannelFutureListener* channel.writeAndFlush(region).addListener(  **new** ChannelFutureListener() {  @Override  **public void** operationComplete(ChannelFuture future)  **throws** Exception {  *// 处理失败* **if** (!future.isSuccess()) {  Throwable cause = future.cause();  *// Do something* }  }  });  } } |

这个示例只适用于文件内容的直接传输，不包括应用程序对数据的任何处理。在需要将数据从文件系统复制到用户内存中时，可以使用ChunkedWriteHandler，它支持异步写大型数据流，而又不会导致大量的内存消耗。

关键是interface ChunkedInput<B>，其中类型参数B 是readChunk()方法返回的类型。Netty 预置了该接口的4 个实现，如表11-7 中所列出的。每个都代表了一个将由ChunkedWriteHandler处理的不定长度的数据流。

代码清单11-12 说明了ChunkedStream 的用法，它是实践中最常用的实现。所示的类使用了一个File 以及一个SslContext 进行实例化。当initChannel()方法被调用时，它将使用所示的ChannelHandler 链初始化该Channel。



当Channel 的状态变为活动的时，WriteStreamHandler 将会逐块地把来自文件中的数据作为ChunkedStream 写入。数据在传输之前将会由SslHandler 加密。

|  |
| --- |
| *// 代码清单11-12 使用ChunkedStream 传输文件内容*  **public class** ChunkedWriteHandlerInitializer  **extends** ChannelInitializer<Channel> {  **private final** File **file**;  **private final** SslContext **sslCtx**;   **public** ChunkedWriteHandlerInitializer(File file, SslContext sslCtx) {  **this**.**file** = file;  **this**.**sslCtx** = sslCtx;  }   @Override  **protected void** initChannel(Channel ch) **throws** Exception {  ChannelPipeline pipeline = ch.pipeline();  *// 将SslHandler 添加到ChannelPipeline 中* pipeline.addLast(**new** SslHandler(**sslCtx**.newEngine(ch.alloc())));  *// 添加ChunkedWriteHandler以处理作为ChunkedInput传入的数据* pipeline.addLast(**new** ChunkedWriteHandler());  *// 一旦连接建立，WriteStreamHandler就开始写文件数据* pipeline.addLast(**new** WriteStreamHandler());  }   **public final class** WriteStreamHandler **extends** ChannelInboundHandlerAdapter {   *// 当连接建立时，channelActive()方法将使用ChunkedInput写文件数据* @Override  **public void** channelActive(ChannelHandlerContext ctx) **throws** Exception {  **super**.channelActive(ctx);  ctx.writeAndFlush(**new** ChunkedStream(**new** FileInputStream(**file**)));  }  } } |

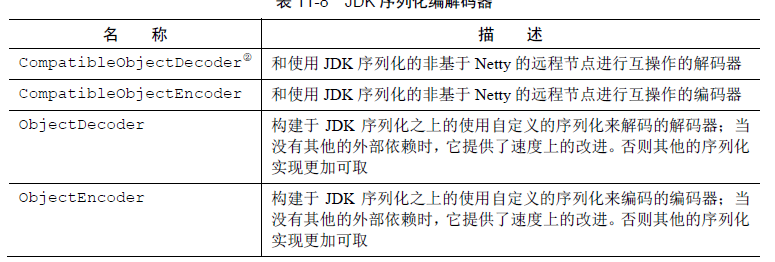
逐块输入 要使用你自己的ChunkedInput 实现，请在ChannelPipeline 中安装一个ChunkedWriteHandler。在本节中，我们讨论了如何通过使用零拷贝特性来高效地传输文件，以及如何通过使用ChunkedWriteHandler 来写大型数据而又不必冒着导致OutOfMemoryError 的风险。在下一节中，我们将仔细研究几种序列化POJO 的方法。

## 11.6 序列化数据

JDK 提供了ObjectOutputStream 和ObjectInputStream，用于通过网络对POJO 的基本数据类型和图进行序列化和反序列化。该API 并不复杂，而且可以被应用于任何实现了java.io.Serializable 接口的对象。但是它的性能也不是非常高效的。在这一节中，我们将看到Netty 必须为此提供什么。

### 11.6.1 JDK 序列化

如果你的应用程序必须要和使用了ObjectOutputStream和ObjectInputStream的远程节点交互，并且兼容性也是你最关心的，那么JDK序列化将是正确的选择。表11-8 中列出了Netty提供的用于和JDK进行互操作的序列化类。



这个类已经在Netty 3.1 中废弃，并不存在于Netty 4.x 中。

### 11.6.2 使用JBoss Marshalling 进行序列化

如果你可以自由地使用外部依赖，那么JBoss Marshalling将是个理想的选择：它比JDK序列化最多快3 倍，而且也更加紧凑。在JBoss Marshalling官方网站主页上的概述中对它是这么定义的：

JBoss Marshalling 是一种可选的序列化API，它修复了在JDK 序列化API 中所发现的许多问题，同时保留了与java.io.Serializable 及其相关类的兼容性，并添加了几个新的可调优参数以及额外的特性，所有的这些都是可以通过工厂配置（如外部序列化器、类/实例查找表、类解析以及对象替换等）实现可插拔的。

Netty 通过表11-9 所示的两组解码器/编码器对为Boss Marshalling 提供了支持。第一组兼容只使用JDK 序列化的远程节点。第二组提供了最大的性能，适用于和使用JBoss Marshalling 的远程节点一起使用。



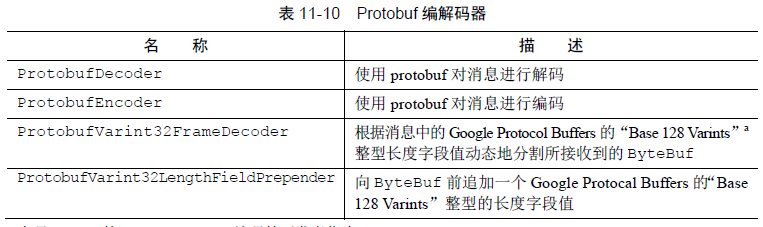
代码清单11-13 展示了如何使用MarshallingDecoder 和MarshallingEncoder。同样，几乎只是适当地配置ChannelPipeline 罢了。

|  |
| --- |
| **// 代码清单11-13 使用JBoss Marshalling**  **public class** MarshallingInitializer **extends** ChannelInitializer<Channel> {  **private final** MarshallerProvider **marshallerProvider**;  **private final** UnmarshallerProvider **unmarshallerProvider**;   **public** MarshallingInitializer(  UnmarshallerProvider unmarshallerProvider,  MarshallerProvider marshallerProvider) {  **this**.**marshallerProvider** = marshallerProvider;  **this**.**unmarshallerProvider** = unmarshallerProvider;  }   @Override  **protected void** initChannel(Channel channel) **throws** Exception {  ChannelPipeline pipeline = channel.pipeline();  *// 添加MarshallingDecoder 以将ByteBuf转换为POJO* pipeline.addLast(**new** MarshallingDecoder(**unmarshallerProvider**));  *// 添加MarshallingEncoder 以将POJO转换为ByteBuf* pipeline.addLast(**new** MarshallingEncoder(**marshallerProvider**));  *// 添加ObjectHandler，以处理普通的实现了Serializable接口的POJO* pipeline.addLast(**new** ObjectHandler());  }   **public static final class** ObjectHandler **extends** SimpleChannelInboundHandler<Serializable> {  @Override  **public void** channelRead0(  ChannelHandlerContext channelHandlerContext,  Serializable serializable) **throws** Exception {  *// Do something* }  } } |

### 11.6.3 通过Protocol Buffers 序列化

Netty序列化的最后一个解决方案是利用Protocol Buffers的编解码器，它是一种由Google公司开发的、现在已经开源的数据交换格式。可以在https://github.com/google/protobuf找到源代码。

Protocol Buffers 以一种紧凑而高效的方式对结构化的数据进行编码以及解码。它具有许多的编程语言绑定，使得它很适合跨语言的项目。表11-10 展示了Netty 为支持protobuf 所提供的ChannelHandler 实现。



a.参见Google 的Protocol Buffers 编码的开发者指南：<https://developers.google.com/protocol-buffers/docs/encoding>。

在这里我们又看到了，使用protobuf 只不过是将正确的ChannelHandler 添加到ChannelPipeline 中，如代码清单11-14 所示。

|  |
| --- |
| **// 代码清单11-14 使用protobuf**  **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 {  ChannelPipeline pipeline = ch.pipeline();  *// 添加ProtobufVarint32FrameDecoder以分隔帧* pipeline.addLast(**new** ProtobufVarint32FrameDecoder());  *// 添加ProtobufEncoder以处理消息的编码* pipeline.addLast(**new** ProtobufEncoder());  *// 添加ProtobufDecoder以解码消息* pipeline.addLast(**new** ProtobufDecoder(**lite**));  *// 添加ObjectHandler 以处理解码消息* pipeline.addLast(**new** ObjectHandler());  }   **public static final class** ObjectHandler  **extends** SimpleChannelInboundHandler<Object> {  @Override  **public void** channelRead0(ChannelHandlerContext ctx, Object msg)  **throws** Exception {  *// Do something with the object* }  } } |

在这一节中，我们探讨了由Netty 专门的解码器和编码器所支持的不同的序列化选项：标准JDK 序列化、JBoss Marshalling 以及Google 的Protocol Buffers。