# 【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。

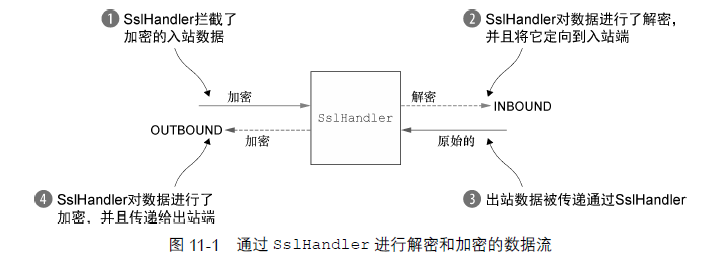
**Netty 的OpenSSL/SSLEngine 实现**

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。



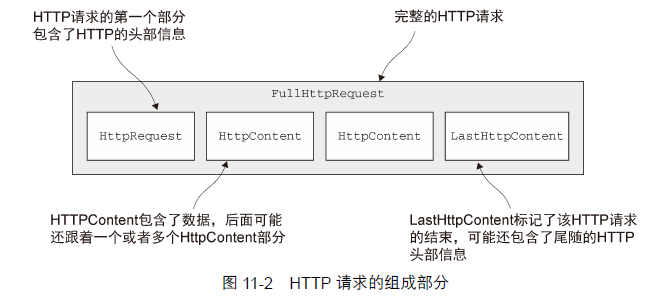
## 11.2 构建基于Netty 的HTTP/HTTPS 应用程序

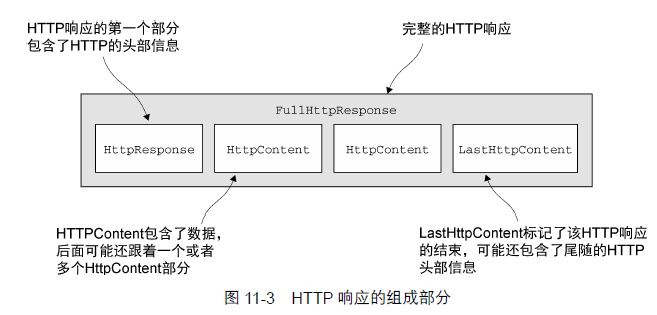
HTTP/HTTPS 是最常见的协议套件之一，并且随着智能手机的成功，它的应用也日益广泛，因为对于任何公司来说，拥有一个可以被移动设备访问的网站几乎是必须的。这些协议也被用于其他方面。许多组织导出的用于和他们的商业合作伙伴通信的WebService API 一般也是基于HTTP（S）的。

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

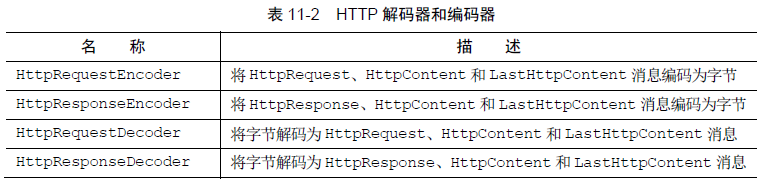
HTTP 是基于请求/响应模式的：客户端向服务器发送一个HTTP 请求，然后服务器将会返回一个HTTP 响应。Netty 提供了多种编码器和解码器以简化对这个协议的使用。图11-2 和图11-3分别展示了生产和消费HTTP 请求和HTTP 响应的方法。





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**));  } } |