第二部分

编解码器

网络只将数据看作是原始的字节序列。然而，我们的应用程序则会把这些字节组织成有意义的信息。在数据和网络字节流之间做相互转换是最常见的编程任务之一。例如，你可能需要处理标准的格式或者协议（如 FTP 或 Telnet）、实现一种由第三方定义的专有二进制协议，或者扩展一种由自己的组织创建的遗留的消息格式。

将应用程序的数据转换为网络格式，以及将网络格式转换为应用程序的数据的组件分别叫作编码器和解码器，同时具有这两种功能的单一组件叫作编解码器。Netty 提供了一系列用来创建所有这些编码器、解码器以及编解码器的工具，从专门为知名协议（如 HTTP 以及 Base64）预构建的类，到你可以按需定制的通用的消息转换编解码器，应有尽有。

第 10 章介绍了编码器和解码器。通过学习一些典型的用例，你将学习到 Netty 的基本的编解码器类。当学习这些类是如何融入整体框架的时候，你将会发现构建它们的 API 和你学过的那些 API 一样，所以你马上就能使用它们。

在第 11 章中，将探索一些 Netty 为处理一些更加专业的场景所提供的编码器和解码器。关于 WebSocket 的那一节是最有意思的，同时它也将为第三部分中关于高级网络协议的详细讨论做好准备。

第 10 章　编解码器框架

本章主要内容

- 解码器、编码器以及编解码器的概述
- Netty 的编解码器类

就像很多标准的架构模式都被各种专用框架所支持一样，常见的数据处理模式往往也是目标实现的很好的候选对象，它可以节省开发人员大量的时间和精力。

当然这也适应于本章的主题：编码和解码，或者数据从一种特定协议的格式到另一种格式的转换。这些任务将由通常称为编解码器的组件来处理。Netty 提供了多种组件，简化了为了支持广泛的协议而创建自定义的编解码器的过程。例如，如果你正在构建一个基于 Netty 的邮件服务器，那么你将会发现 Netty 对于编解码器的支持对于实现 POP3、IMAP 和 SMTP 协议来说是多么的宝贵。

10.1　什么是编解码器

每个网络应用程序都必须定义如何解析在两个节点之间来回传输的原始字节，以及如何将其和目标应用程序的数据格式做相互转换。这种转换逻辑由编解码器处理，编解码器由编码器和解码器组成，它们每种都可以将字节流从一种格式转换为另一种格式。那么它们的区别是什么呢？

如果将消息看作是对于特定的应用程序具有具体含义的结构化的字节序列——它的数据。那么编码器是将消息转换为适合于传输的格式（最有可能的就是字节流）；而对应的解码器则是将网络字节流转换回应用程序的消息格式。因此，编码器操作出站数据，而解码器处理入站数据。

记住这些背景信息，接下来让我们研究一下 Netty 所提供的用于实现这两种组件的类。

10.2　解码器

在这一节中，我们将研究 Netty 所提供的解码器类，并提供关于何时以及如何使用它们的具体示例。这些类覆盖了两个不同的用例：

- 将字节解码为消息——ByteToMessageDecoder 和 ReplayingDecoder；
- 将一种消息类型解码为另一种——MessageToMessageDecoder。

因为解码器是负责将入站数据从一种格式转换到另一种格式的，所以知道 Netty 的解码器实现了 ChannelInboundHandler 也不会让你感到意外。

什么时候会用到解码器呢？很简单：每当需要为 ChannelPipeline 中的下一个 ChannelInboundHandler 转换入站数据时会用到。此外，得益于 ChannelPipeline 的设计，可以将多个解码器链接在一起，以实现任意复杂的转换逻辑，这也是 Netty 是如何支持代码的模块化以及复用的一个很好的例子。

10.2.1 抽象类 ByteToMessageDecoder

将字节解码为消息（或者另一个字节序列）是一项如此常见的任务，以至于 Netty 为它提供了一个抽象的基类：ByteToMessageDecoder。由于你不可能知道远程节点是否会一次性地发送一个完整的消息，所以这个类会对入站数据进行缓冲，直到它准备好处理。表 10-1 解释了它最重要的两个方法。

表 10-1 ByteToMessageDecoder API

方 法	描 述
decode(ChannelHandlerContext ctx, ByteBuf in, List<Object> out)	这是你必须实现的唯一抽象方法。decode()方法被调用时将会传入一个包含了传入数据的 ByteBuf，以及一个用来添加解码消息的 List。对这个方法的调用将会重复进行，直到确定没有新的元素被添加到该 List，或者该 ByteBuf 中没有更多可读取的字节时为止。然后，如果该 List 不为空，那么它的内容将会被传递给 ChannelPipeline 中的下一个 ChannelInboundHandler
decodeLast(ChannelHandlerContext ctx, ByteBuf in, List<Object> out)	Netty提供的这个默认实现只是简单地调用了decode()方法。当Channel的状态变为非活动时，这个方法将会被调用一次。可以重写该方法以提供特殊的处理[①]

下面举一个如何使用这个类的示例，假设你接收了一个包含简单 int 的字节流，每个 int 都需要被单独处理。在这种情况下，你需要从入站 ByteBuf 中读取每个 int，并将它传递给 ChannelPipeline 中的下一个 ChannelInboundHandler。为了解码这个字节流，你要扩展 ByteToMessageDecoder 类。（需要注意的是，原子类型的 int 在被添加到 List 中时，会被自动装箱为 Integer。）

该设计如图 10-1 所示。

每次从入站 ByteBuf 中读取 4 字节，将其解码为一个 int，然后将它添加到一个 List 中。当没有更多的元素可以被添加到该 List 中时，它的内容将会被发送给下一个 ChannelInboundHandler。

① 比如用来产生一个 LastHttpContent 消息。——译者注

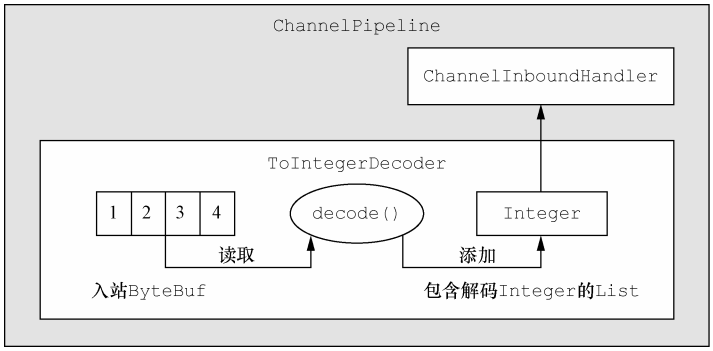


图 10-1 ToIntegerDecoder

代码清单 10-1 展示了 ToIntegerDecoder 的代码。

代码清单 10-1 ToIntegerDecoder 类扩展了 ByteToMessageDecoder

```
public class ToIntegerDecoder extends ByteToMessageDecoder {
    @Override
    public void decode(ChannelHandlerContext ctx, ByteBuf in,
        List<Object> out) throws Exception {
        if (in.readableBytes() >= 4) {
            out.add(in.readInt());
        }
    }
}
```

扩展 ByteToMessageDecoder 类，以将字节解码为特定的格式

检查是否至少有 4 字节可读（一个 int 的字节长度）

从入站 ByteBuf 中读取一个 int，并将其添加到解码消息的 List 中

虽然 ByteToMessageDecoder 使得可以很简单地实现这种模式，但是你可能会发现，在调用 readInt() 方法前不得不验证所输入的 ByteBuf 是否具有足够的数据有点繁琐。在下一节中，我们将讨论 ReplayingDecoder，它是一个特殊的解码器，以少量的开销消除了这个步骤。

编解码器中的引用计数

正如我们在第 5 章和第 6 章中所提到的，引用计数需要特别的注意。对于编码器和解码器来说，其过程也是相当的简单：一旦消息被编码或者解码，它就会被 ReferenceCountUtil.release(message) 调用自动释放。如果你需要保留引用以便稍后使用，那么你可以调用 ReferenceCountUtil.retain(message) 方法。这将会增加该引用计数，从而防止该消息被释放。

10.2.2 抽象类 ReplayingDecoder

ReplayingDecoder 扩展了 ByteToMessageDecoder 类（如代码清单 10-1 所示），使得我们不必调用 readableBytes() 方法。它通过使用一个自定义的 ByteBuf 实现，ReplayingDecoderByteBuf，包装传入的 ByteBuf 实现了这一点，其将在内部执行该调用[①]。

① 指调用 readableBytes() 方法。——译者注

这个类的完整声明是：

```
public abstract class ReplayingDecoder<S> extends ByteToMessageDecoder
```

类型参数 S 指定了用于状态管理的类型，其中 Void 代表不需要状态管理。代码清单 10-2 展示了基于 ReplayingDecoder 重新实现的 ToIntegerDecoder。

代码清单 10-2 ToIntegerDecoder2 类扩展了 ReplayingDecoder

```
public class ToIntegerDecoder2 extends ReplayingDecoder<Void> {
    @Override
    public void decode(ChannelHandlerContext ctx, ByteBuf in,
        List<Object> out) throws Exception {
        out.add(in.readInt());
    }
}
```

扩展 ReplayingDecoder<Void>以将字节解码为消息

传入的 ByteBuf 是 ReplayingDecoderByteBuf

从入站 ByteBuf 中读取一个 int，并将其添加到解码消息的 List 中

和之前一样，从ByteBuf中提取的int将会被添加到List中。如果没有足够的字节可用，这个readInt()方法的实现将会抛出一个Error[①]，其将在基类中被捕获并处理。当有更多的数据可供读取时，该decode()方法将会被再次调用。（参见表 10-1 中关于decode()方法的描述。）

请注意 ReplayingDecoderByteBuf 的下面这些方面：

- 并不是所有的 ByteBuf 操作都被支持，如果调用了一个不被支持的方法，将会抛出一个 UnsupportedOperationException；
- ReplayingDecoder 稍慢于 ByteToMessageDecoder。

如果对比代码清单 10-1 和代码清单 10-2，你会发现后者明显更简单。示例本身是很基本的，所以请记住，在真实的、更加复杂的情况下，使用一种或者另一种作为基类所带来的差异可能是很显著的。这里有一个简单的准则：如果使用 ByteToMessageDecoder 不会引入太多的复杂性，那么请使用它；否则，请使用 ReplayingDecoder。

更多的解码器

下面的这些类处理更加复杂的用例：

- io.netty.handler.codec.LineBasedFrameDecoder——这个类在 Netty 内部也有使用，它使用了行尾控制字符（\n 或者\r\n）来解析消息数据；
- io.netty.handler.codec.http.HttpObjectDecoder——一个 HTTP 数据的解码器。

在 io.netty.handler.codec 子包下面，你将会发现更多用于特定用例的编码器和解码器实现。更多有关信息参见 Netty 的 Javadoc。

10.2.3 抽象类 MessageToMessageDecoder

在这一节中，我们将解释如何使用下面的抽象基类在两个消息格式之间进行转换（例如，从

① 这里实际上抛出的是一个 Signal，详见 io.netty.util.Signal 类。——译者注

一种 POJO 类型转换为另一种):

```
public abstract class MessageToMessageDecoder<I>
    extends ChannelInboundHandlerAdapter
```

类型参数 I 指定了 decode() 方法的输入参数 msg 的类型，它是你必须实现的唯一方法。表 10-2 展示了这个方法的详细信息。

表 10-2 MessageToMessageDecoder API

方 法	描 述
decode(ChannelHandlerContext ctx, I msg, List<Object> out)	对于每个需要被解码为另一种格式的入站消息来说，该方法都将会被调用。解码消息随后会被传递给 ChannelPipeline 中的下一个 ChannelInboundHandler

在这个示例中，我们将编写一个 IntegerToStringDecoder 解码器来扩展 MessageToMessageDecoder<Integer>。它的 decode() 方法会把 Integer 参数转换为它的 String 表示，并将拥有下列签名：

```
public void decode( ChannelHandlerContext ctx,
    Integer msg, List<Object> out ) throws Exception
```

和之前一样，解码的 String 将被添加到传出的 List 中，并转发给下一个 ChannelInboundHandler。

该设计如图 10-2 所示。

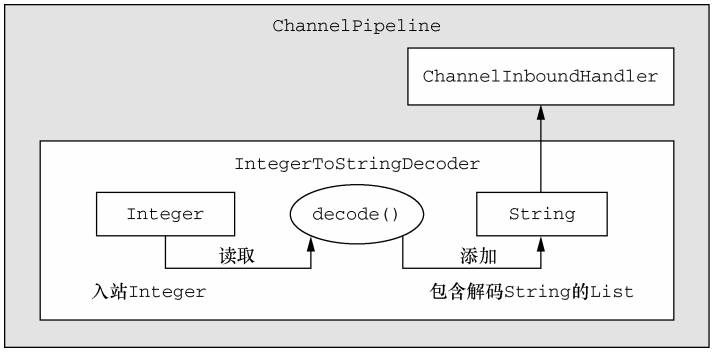


图 10-2 IntegerToStringDecoder

代码清单 10-3 给出了 IntegerToStringDecoder 的实现。

代码清单 10-3 IntegerToStringDecoder 类

```
public class IntegerToStringDecoder extends
    MessageToMessageDecoder<Integer> {
    @Override
    public void decode(ChannelHandlerContext ctx, Integer msg
        List<Object> out) throws Exception {
```

扩展了 MessageToMessageDecoder<Integer>

```
            out.add(String.valueOf(msg));
        }
    }
```

将 Integer 消息转换为它的 String 表示，并将其添加到输出的 List 中

HttpObjectAggregator

有关更加复杂的例子，请研究 io.netty.handler.codec.http.HttpObjectAggregator 类，它扩展了 MessageToMessageDecoder<HttpObject>。

10.2.4 TooLongFrameException 类

由于 Netty 是一个异步框架，所以需要在字节可以解码之前在内存中缓冲它们。因此，不能让解码器缓冲大量的数据以至于耗尽可用的内存。为了解除这个常见的顾虑，Netty 提供了 TooLongFrameException 类，其将由解码器在帧超出指定的大小限制时抛出。

为了避免这种情况，你可以设置一个最大字节数的阈值，如果超出该阈值，则会导致抛出一个 TooLongFrameException（随后会被 ChannelHandler.exceptionCaught()方法捕获）。然后，如何处理该异常则完全取决于该解码器的用户。某些协议（如 HTTP）可能允许你返回一个特殊的响应。而在其他的情况下，唯一的选择可能就是关闭对应的连接。

代码清单 10-4 展示了 ByteToMessageDecoder 是如何使用 TooLongFrameException 来通知 ChannelPipeline 中的其他 ChannelHandler 发生了帧大小溢出的。需要注意的是，如果你正在使用一个可变帧大小的协议，那么这种保护措施将是尤为重要的。

代码清单 10-4 TooLongFrameException

扩展 ByteToMessageDecoder 以将字节解码为消息

```
public class SafeByteToMessageDecoder extends ByteToMessageDecoder {
    private static final int MAX_FRAME_SIZE = 1024;
    @Override
    public void decode(ChannelHandlerContext ctx, ByteBuf in,
        List<Object> out) throws Exception {
            int readable = in.readableBytes();
            if (readable > MAX_FRAME_SIZE) {
                in.skipBytes(readable);
                throw new TooLongFrameException("Frame too big!");
            }
            // do something
            ...
    }
}
```

检查缓冲区中是否有超过 MAX_FRAME_SIZE 个字节

跳过所有的可读字节，抛出 TooLongFrameException 并通知 ChannelHandler

到目前为止，我们已经探讨了解码器的常规用例，以及 Netty 所提供的用于构建它们的抽象基类。但是解码器只是硬币的一面。硬币的另一面是编码器，它将消息转换为适合于传出传输的格式。这些编码器完备了编解码器 API，它们将是我们的下一个主题。

10.3 编码器

回顾一下我们先前的定义，编码器实现了 ChannelOutboundHandler，并将出站数据从一种格式转换为另一种格式，和我们方才学习的解码器的功能正好相反。Netty 提供了一组类，用于帮助你编写具有以下功能的编码器：

- 将消息编码为字节；
- 将消息编码为消息[①]。

我们将首先从抽象基类 MessageToByteEncoder 开始来对这些类进行考察。

10.3.1 抽象类 MessageToByteEncoder

前面我们看到了如何使用 ByteToMessageDecoder 来将字节转换为消息。现在我们将使用 MessageToByteEncoder 来做逆向的事情。表 10-3 展示了该 API。

表 10-3 **MessageToByteEncoder** API

方　法	描　述
encode(ChannelHandlerContext ctx, I msg, ByteBuf out)	encode()方法是你需要实现的唯一抽象方法。它被调用时将会传入要被该类编码为 ByteBuf 的（类型为 I 的）出站消息。该 ByteBuf 随后将会被转发给 ChannelPipeline 中的下一个 ChannelOutboundHandler

你可能已经注意到了，这个类只有一个方法，而解码器有两个。原因是解码器通常需要在 Channel 关闭之后产生最后一个消息（因此也就有了 decodeLast()方法）。这显然不适用于编码器的场景——在连接被关闭之后仍然产生一个消息是毫无意义的。

图 10-3 展示了 ShortToByteEncoder，其接受一个 Short 类型的实例作为消息，将它编码为 Short 的原子类型值，并将它写入 ByteBuf 中，其将随后被转发给 ChannelPipeline 中的下一个 ChannelOutboundHandler。每个传出的 Short 值都将会占用 ByteBuf 中的 2 字节。

ShortToByteEncoder 的实现如代码清单 10-5 所示。

代码清单 10-5 ShortToByteEncoder 类

```
public class ShortToByteEncoder extends MessageToByteEncoder<Short> {
    @Override
    public void encode(ChannelHandlerContext ctx, Short msg, ByteBuf out)
        throws Exception {
        out.writeShort(msg);
    }
}
```

扩展了 MessageToByteEncoder

将 Short 写入 ByteBuf 中

① 另外一种格式的消息。——译者注

Netty 提供了一些专门化的 MessageToByteEncoder，你可以基于它们实现自己的编码器。WebSocket08FrameEncoder 类提供了一个很好的实例。你可以在 io.netty.handler.codec.http.websocketx 包中找到它。

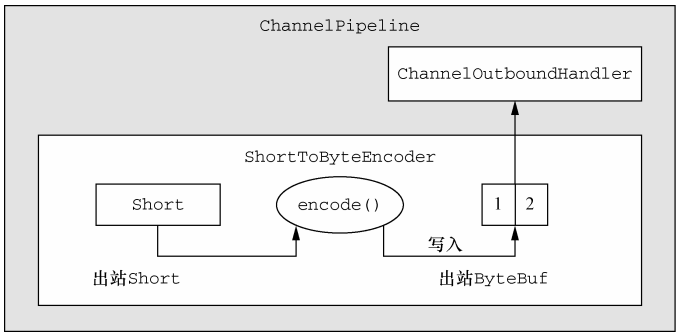


图 10-3 ShortToByteEncoder

10.3.2 抽象类 MessageToMessageEncoder

你已经看到了如何将入站数据从一种消息格式解码为另一种。为了完善这幅图，我们将展示对于出站数据将如何从一种消息编码为另一种。MessageToMessageEncoder 类的 encode() 方法提供了这种能力，如表 10-4 所示。

表 10-4 **MessageToMessageEncoder** API

名 称	描 述
encode(ChannelHandlerContext ctx, I msg, List<Object> out)	这是你需要实现的唯一方法。每个通过 write() 方法写入的消息都将会被传递给 encode() 方法，以编码为一个或者多个出站消息。随后，这些出站消息将会被转发给 ChannelPipeline 中的下一个 ChannelOutboundHandler

为了演示，代码清单 10-6 使用 IntegerToStringEncoder 扩展了 MessageToMessageEncoder。其设计如图 10-4 所示。

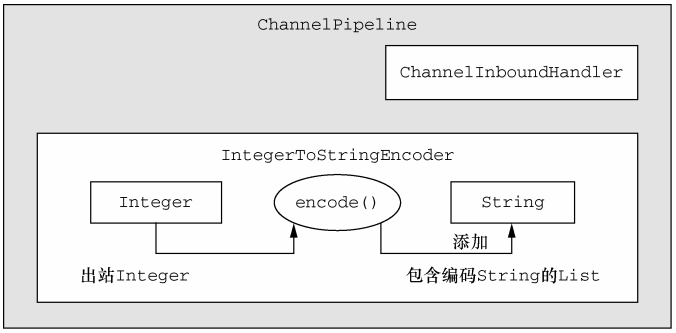


图 10-4 IntegerToStringEncoder

如代码清单 10-6 所示，编码器将每个出站 Integer 的 String 表示添加到了该 List 中。

代码清单 10-6 IntegerToStringEncoder 类

```
public class IntegerToStringEncoder
    extends MessageToMessageEncoder<Integer> {
    @Override
    public void encode(ChannelHandlerContext ctx, Integer msg
        List<Object> out) throws Exception {
        out.add(String.valueOf(msg));
    }
}
```

扩展了 MessageToMessageEncoder

将 Integer 转换为 String，并将其添加到 List 中

关于有趣的 MessageToMessageEncoder 的专业用法，请查看 io.netty.handler.codec.protobuf.ProtobufEncoder 类，它处理了由 Google 的 Protocol Buffers 规范所定义的数据格式。

10.4 抽象的编解码器类

虽然我们一直将解码器和编码器作为单独的实体讨论，但是你有时将会发现在同一个类中管理入站和出站数据和消息的转换是很有用的。Netty 的抽象编解码器类正好用于这个目的，因为它们每个都将捆绑一个解码器/编码器对，以处理我们一直在学习的这两种类型的操作。正如同你可能已经猜想到的，这些类同时实现了 ChannelInboundHandler 和 ChannelOutboundHandler 接口。

为什么我们并没有一直优先于单独的解码器和编码器使用这些复合类呢？因为通过尽可能地将这两种功能分开，最大化了代码的可重用性和可扩展性，这是 Netty 设计的一个基本原则。

在我们查看这些抽象的编解码器类时，我们将会把它们与相应的单独的解码器和编码器进行比较和参照。

10.4.1 抽象类 ByteToMessageCodec

让我们来研究这样的一个场景：我们需要将字节解码为某种形式的消息，可能是 POJO，随后再次对它进行编码。ByteToMessageCodec 将为我们处理好这一切，因为它结合了 ByteToMessageDecoder 以及它的逆向——MessageToByteEncoder。表 10-5 列出了其中重要的方法。

任何的请求/响应协议都可以作为使用ByteToMessageCodec的理想选择。例如，在某个SMTP的实现中，编解码器将读取传入字节，并将它们解码为一个自定义的消息类型，如 SmtpRequest[①]。而在接收端，当一个响应被创建时，将会产生一个SmtpResponse，其将被编码回字节以便进行传输。

① 位于基于 Netty 的 SMTP/LMTP 客户端项目中（https://github.com/normanmaurer/niosmtp）。——译者注

表 10-5 ByteToMessageCodec API

方法名称	描述
decode(ChannelHandlerContext ctx, ByteBuf in, List<Object>)	只要有字节可以被消费，这个方法就将会被调用。它将入站 ByteBuf 转换为指定的消息格式，并将其转发给 ChannelPipeline 中的下一个 ChannelInboundHandler
decodeLast(ChannelHandlerContext ctx, ByteBuf in, List<Object> out)	这个方法的默认实现委托给了 decode()方法。它只会在 Channel 的状态变为非活动时被调用一次。它可以被重写以实现特殊的处理
encode(ChannelHandlerContext ctx, I msg, ByteBuf out)	对于每个将被编码并写入出站 ByteBuf 的（类型为 I 的）消息来说，这个方法都将会被调用

10.4.2 抽象类 MessageToMessageCodec

在 10.3.1 节中，你看到了一个扩展了 MessageToMessageEncoder 以将一种消息格式转换为另外一种消息格式的例子。通过使用 MessageToMessageCodec，我们可以在一个单个的类中实现该转换的往返过程。MessageToMessageCodec 是一个参数化的类，定义如下：

```
public abstract class MessageToMessageCodec<INBOUND_IN,OUTBOUND_IN>
```

表 10-6 列出了其中重要的方法。

表 10-6 MessageToMessageCodec 的方法

方法名称	描述
protected abstract decode(ChannelHandlerContext ctx, INBOUND_IN msg, List<Object> out)	这个方法被调用时会被传入 INBOUND_IN 类型的消息。它将把它们解码为 OUTBOUND_IN 类型的消息，这些消息将被转发给 ChannelPipeline 中的下一个 ChannelInboundHandler
protected abstract encode(ChannelHandlerContext ctx, OUTBOUND_IN msg, List<Object> out)	对于每个 OUTBOUND_IN 类型的消息，这个方法都将会被调用。这些消息将会被编码为 INBOUND_IN 类型的消息，然后被转发给 ChannelPipeline 中的下一个 ChannelOutboundHandler

decode()方法是将 INBOUND_IN 类型的消息转换为 OUTBOUND_IN 类型的消息，而 encode()方法则进行它的逆向操作。将 INBOUND_IN 类型的消息看作是通过网络发送的类型，而将 OUTBOUND_IN 类型的消息看作是应用程序所处理的类型，将可能有所裨益[①]。

① 即有助于理解这两个类型签名的实际意义。——译者注

虽然这个编解码器可能看起来有点高深，但是它所处理的用例却是相当常见的：在两种不同的消息 API 之间来回转换数据。当我们不得不和使用遗留或者专有消息格式的 API 进行互操作时，我们经常会遇到这种模式。

WebSocket 协议

下面关于 MessageToMessageCodec 的示例引用了一个新出的 WebSocket 协议，这个协议能实现 Web 浏览器和服务器之间的全双向通信。我们将在第 12 章中详细地讨论 Netty 对于 WebSocket 的支持。

代码清单 10-7 展示了这样的对话[①]可能的实现方式。我们的WebSocketConvertHandler在参数化MessageToMessageCodec时将使用INBOUND_IN类型的WebSocketFrame，以及OUTBOUND_IN类型的MyWebSocketFrame，后者是WebSocketConvertHandler本身的一个静态嵌套类。

代码清单 10-7 使用 MessageToMessageCodec

```
public class WebSocketConvertHandler extends
    MessageToMessageCodec<WebSocketFrame,
    WebSocketConvertHandler.MyWebSocketFrame> {
    @Override
    protected void encode(ChannelHandlerContext ctx,
        WebSocketConvertHandler.MyWebSocketFrame msg,
        List<Object> out) throws Exception {
        ByteBuf payload = msg.getData().duplicate().retain();
        switch (msg.getType()) {
            case BINARY:
                out.add(new BinaryWebSocketFrame(payload));
                break;
            case TEXT:
                out.add(new TextWebSocketFrame(payload));
                break;
            case CLOSE:
                out.add(new CloseWebSocketFrame(true, 0, payload));
                break;
            case CONTINUATION:
                out.add(new ContinuationWebSocketFrame(payload));
                break;
            case PONG:
                out.add(new PongWebSocketFrame(payload));
                break;
            case PING:
                out.add(new PingWebSocketFrame(payload));
                break;
            default:
                throw new IllegalStateException(
                    "Unsupported websocket msg " + msg);
        }
```

将 MyWebSocketFrame 编码为指定的 WebSocketFrame 子类型

实例化一个指定子类型的 WebSocketFrame

① 指 Web 浏览器和服务器之间的双向通信。——译者注

```
}

@Override
protected void decode(ChannelHandlerContext ctx, WebSocketFrame msg,
    List<Object> out) throws Exception {
        ByteBuf payload = msg.content().duplicate().retain();
        if (msg instanceof BinaryWebSocketFrame) {
            out.add(new MyWebSocketFrame(
                MyWebSocketFrame.FrameType.BINARY, payload));
        } else
        if (msg instanceof CloseWebSocketFrame) {
            out.add(new MyWebSocketFrame (
                MyWebSocketFrame.FrameType.CLOSE, payload));
        } else
        if (msg instanceof PingWebSocketFrame) {
            out.add(new MyWebSocketFrame (
                MyWebSocketFrame.FrameType.PING, payload));
        } else
        if (msg instanceof PongWebSocketFrame) {
            out.add(new MyWebSocketFrame (
                MyWebSocketFrame.FrameType.PONG, payload));
        } else
        if (msg instanceof TextWebSocketFrame) {
            out.add(new MyWebSocketFrame (
                MyWebSocketFrame.FrameType.TEXT, payload));
        } else
        if (msg instanceof ContinuationWebSocketFrame) {
            out.add(new MyWebSocketFrame (
                MyWebSocketFrame.FrameType.CONTINUATION, payload));
        } else
        {
            throw new IllegalStateException(
                "Unsupported websocket msg " + msg);
    }
}

public static final class MyWebSocketFrame {
    public enum FrameType {
        BINARY,
        CLOSE,
        PING,
        PONG,
        TEXT,
        CONTINUATION
    }
    private final FrameType type;
    private final ByteBuf data;

    public MyWebSocketFrame(FrameType type, ByteBuf data) {
        this.type = type;
        this.data = data;
    }
```

将 WebSocketFrame 解码为 MyWebSocketFrame，并设置 FrameType

声明 WebSocketConvertHandler 所使用的 OUTBOUND_IN 类型

定义拥有被包装的有效负载的 WebSocketFrame 的类型

```
        public FrameType getType() {
            return type;
        }

        public ByteBuf getData() {
            return data;
        }
    }
}
```

10.4.3 CombinedChannelDuplexHandler 类

正如我们前面所提到的，结合一个解码器和编码器可能会对可重用性造成影响。但是，有一种方法既能够避免这种惩罚，又不会牺牲将一个解码器和一个编码器作为一个单独的单元部署所带来的便利性。CombinedChannelDuplexHandler 提供了这个解决方案，其声明为：

```
public class CombinedChannelDuplexHandler
    <I extends ChannelInboundHandler,
    O extends ChannelOutboundHandler>
```

这个类充当了 ChannelInboundHandler 和 ChannelOutboundHandler（该类的类型参数 I 和 O）的容器。通过提供分别继承了解码器类和编码器类的类型，我们可以实现一个编解码器，而又不必直接扩展抽象的编解码器类。我们将在下面的示例中说明这一点。

首先，让我们研究代码清单 10-8 中的 ByteToCharDecoder。注意，该实现扩展了 ByteToMessageDecoder，因为它要从 ByteBuf 中读取字符。

代码清单 10-8 ByteToCharDecoder 类

```
public class ByteToCharDecoder extends ByteToMessageDecoder {
    @Override
    public void decode(ChannelHandlerContext ctx, ByteBuf in,
        List<Object> out) throws Exception {
            while (in.readableBytes() >= 2) {
                out.add(in.readChar());
            }
    }
}
```

扩展了 ByteToMessageDecoder

将一个或者多个 Character 对象添加到传出的 List 中

这里的 decode() 方法一次将从 ByteBuf 中提取 2 字节，并将它们作为 char 写入到 List 中，其将会被自动装箱为 Character 对象。

代码清单 10-9 包含了 CharToByteEncoder，它能将 Character 转换回字节。这个类扩展了 MessageToByteEncoder，因为它需要将 char 消息编码到 ByteBuf 中。这是通过直接写入 ByteBuf 做到的。

代码清单 10-9 CharToByteEncoder 类

```
public class CharToByteEncoder extends
    MessageToByteEncoder<Character> {
    @Override
    public void encode(ChannelHandlerContext ctx, Character msg,
        ByteBuf out) throws Exception {
        out.writeChar(msg);
    }
}
```

扩展了 MessageToByteEncoder

将 Character 解码为 char，并将其写入到出站 ByteBuf 中

既然我们有了解码器和编码器，我们将会结合它们来构建一个编解码器。代码清单 10-10 展示了这是如何做到的。

代码清单 10-10 CombinedChannelDuplexHandler<I,O>

```
public class CombinedByteCharCodec extends
    CombinedChannelDuplexHandler<ByteToCharDecoder, CharToByteEncoder> {
    public CombinedByteCharCodec() {
        super(new ByteToCharDecoder(), new CharToByteEncoder());
    }
}
```

通过该解码器和编码器实现参数化 CombinedByteCharCodec

将委托实例传递给父类

正如你所能看到的，在某些情况下，通过这种方式结合实现相对于使用编解码器类的方式来说可能更加的简单也更加的灵活。当然，这可能也归结于个人的偏好问题。

10.5 小结

在本章中，我们学习了如何使用 Netty 的编解码器 API 来编写解码器和编码器。你也了解了为什么使用这个 API 相对于直接使用 `ChannelHandler` API 更好。

你看到了抽象的编解码器类是如何为在一个实现中处理解码和编码提供支持的。如果你需要更大的灵活性，或者希望重用现有的实现，那么你还可以选择结合他们，而无需扩展任何抽象的编解码器类。

在下一章中，我们将讨论作为 Netty 框架本身的一部分的 `ChannelHandler` 实现和编解码器，你可以利用它们来处理特定的协议和任务。

第 11 章　预置的 ChannelHandler 和编解码器

本章主要内容

- 通过 SSL/TLS 保护 Netty 应用程序
- 构建基于 Netty 的 HTTP/HTTPS 应用程序
- 处理空闲的连接和超时
- 解码基于分隔符的协议和基于长度的协议
- 写大型数据

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[①]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。

图 11-1 展示了使用 `SslHandler` 的数据流。

① 传输层安全（TLS）协议，1.2 版：http://tools.ietf.org/html/rfc5246。

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。

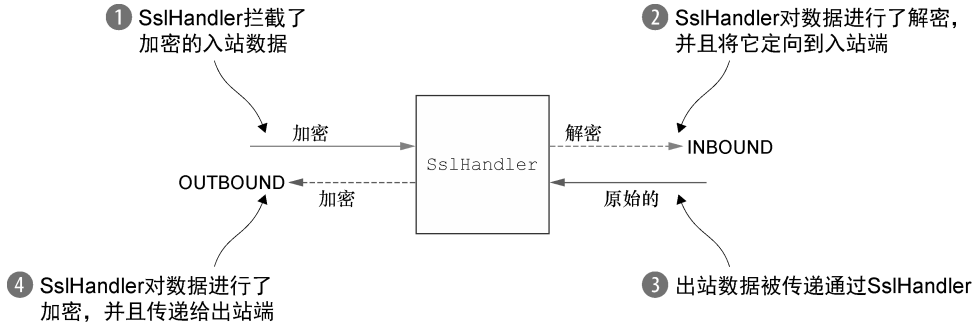


图 11-1　通过 SslHandler 进行解密和加密的数据流

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;

    public SslChannelInitializer(SslContext context,
        boolean startTls) {
        this.context = context;
        this.startTls = startTls;
    }

    @Override
    protected void initChannel(Channel ch) throws Exception {
        SSLEngine engine = context.newEngine(ch.alloc());
        ch.pipeline().addFirst("ssl",
            new SslHandler(engine, 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）

对于每个 SslHandler 实例，都使用 Channel 的 ByteBufAllocator 从 SslContext 获取一个新的 SSLEngine

将 SslHandler 作为第一个 ChannelHandler 添加到 ChannelPipeline 中

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。

表 11-1 SslHandler 的方法

方法名称	描述
setHandshakeTimeout (long,TimeUnit) setHandshakeTimeoutMillis (long) getHandshakeTimeoutMillis()	设置和获取超时时间，超时之后，握手 ChannelFuture 将会被通知失败
setCloseNotifyTimeout (long,TimeUnit) setCloseNotifyTimeoutMillis (long) getCloseNotifyTimeoutMillis()	设置和获取超时时间，超时之后，将会触发一个关闭通知并关闭连接。这也将会导致通知该 ChannelFuture 失败
handshakeFuture()	返回一个在握手完成后将会得到通知的 ChannelFuture。如果握手先前已经执行过了，则返回一个包含了先前的握手结果的 ChannelFuture
close() close(ChannelPromise) close(ChannelHandlerContext,ChannelPromise)	发送 close_notify 以请求关闭并销毁底层的 SslEngine

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。

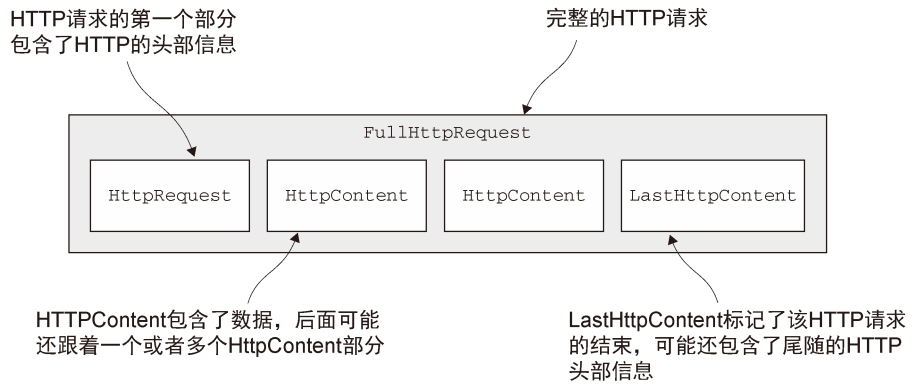


图 11-2 HTTP 请求的组成部分

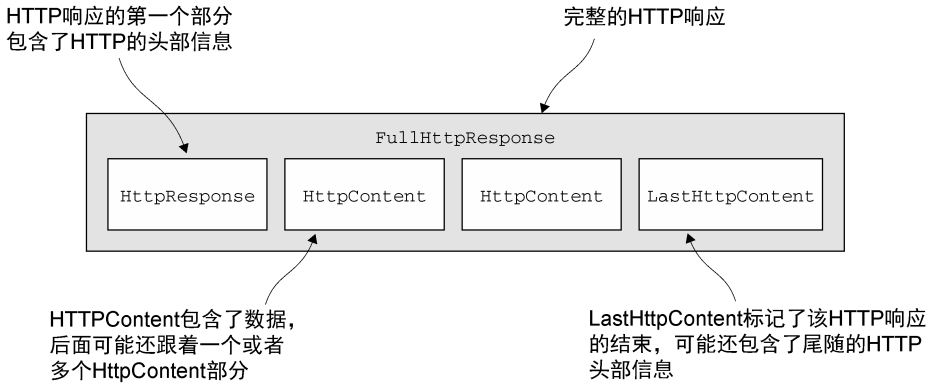


图 11-3 HTTP 响应的组成部分

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。

表 11-2 HTTP 解码器和编码器

名 称	描 述
`HttpRequestEncoder`	将 `HttpRequest`、`HttpContent` 和 `LastHttpContent` 消息编码为字节
`HttpResponseEncoder`	将 `HttpResponse`、`HttpContent` 和 `LastHttpContent` 消息编码为字节
`HttpRequestDecoder`	将字节解码为 `HttpRequest`、`HttpContent` 和 `LastHttpContent` 消息
`HttpResponseDecoder`	将字节解码为 `HttpResponse`、`HttpContent` 和 `LastHttpContent` 消息

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) {
            pipeline.addLast("decoder", new HttpResponseDecoder());
            pipeline.addLast("encoder", new HttpRequestEncoder());
        } else {
            pipeline.addLast("decoder", new HttpRequestDecoder());
            pipeline.addLast("encoder", new HttpResponseEncoder());
        }
    }
}
```

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

如果是服务器，则添加 HttpRequestDecoder 以接收来自客户端的请求

如果是服务器，则添加 HttpResponseEncoder 以向客户端发送响应

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) {
            pipeline.addLast("codec", new HttpClientCodec());
        } else {
            pipeline.addLast("codec", new HttpServerCodec());
        }
        pipeline.addLast("aggregator",
            new HttpObjectAggregator(512 * 1024));
    }
}
```

如果是客户端，则添加 HttpClientCodec

如果是服务器，则添加 HttpServerCodec

将最大的消息大小为 512 KB 的 HttpObjectAggregator 添加到 ChannelPipeline

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) {
            pipeline.addLast("codec", new HttpClientCodec());
            pipeline.addLast("decompressor",
                new HttpContentDecompressor());
        } else {
            pipeline.addLast("codec", new HttpServerCodec());
            pipeline.addLast("compressor",
            new HttpContentCompressor());
        }
    }
}
```

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

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）

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>
</dependency>
```

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());
        pipeline.addFirst("ssl", new SslHandler(engine));

        if (isClient) {
            pipeline.addLast("codec", new HttpClientCodec());
        } else {
            pipeline.addLast("codec", new HttpServerCodec());
        }
    }
}
```

将 SslHandler 添加到 ChannelPipeline 中以使用 HTTPS

如果是客户端，则添加 HttpClientCodec

如果是服务器，则添加 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[①]，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。”[②]

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。

① Comet 就是一个例子：http://en.wikipedia.org/wiki/Comet_%28programming%29。

② RFC 6455，WebSocket 协议，http://tools.ietf.org/html/rfc6455。

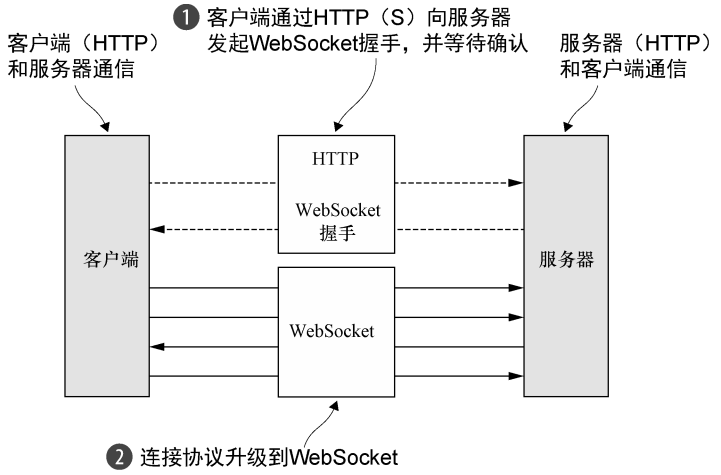


图 11-4 WebSocket 协议

表 11-3 **WebSocketFrame** 类型

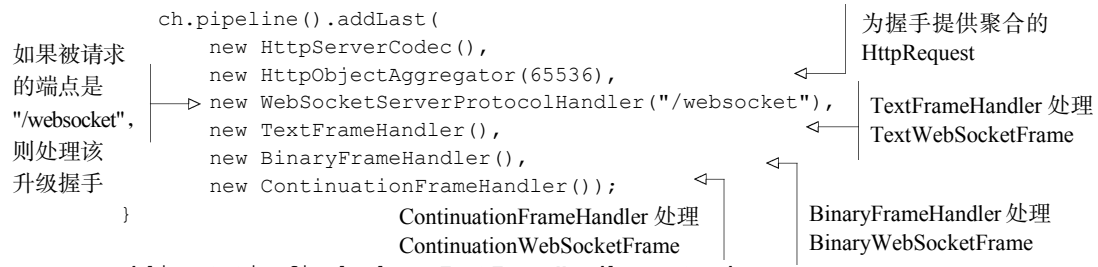
名 称	描 述
BinaryWebSocketFrame	数据帧：二进制数据
TextWebSocketFrame	数据帧：文本数据
ContinuationWebSocketFrame	数据帧:属于上一个 BinaryWebSocketFrame 或者 TextWebSocketFrame 的文本的或者二进制数据
CloseWebSocketFrame	控制帧：一个 CLOSE 请求、关闭的状态码以及关闭的原因
PingWebSocketFrame	控制帧：请求一个 PongWebSocketFrame
PongWebSocketFrame	控制帧：对 PingWebSocketFrame 请求的响应

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[①]。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 {
```

① 关于 WebSocket 的客户端示例，请参考 Netty 源代码中所包含的例子：https://github.com/netty/netty/tree/4.1/example/src/main/java/io/netty/example/http/websocketx/client。




```
            ch.pipeline().addLast(
                new HttpServerCodec(),
                new HttpObjectAggregator(65536),
                new WebSocketServerProtocolHandler("/websocket"),
                new TextFrameHandler(),
                new BinaryFrameHandler(),
                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。

更加全面的示例参见第 12 章，那一章会深入探讨实时 WebSocket 应用程序的设计。

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。

表 11-4 用于空闲连接以及超时的 ChannelHandler

名 称	描 述
IdleStateHandler	当连接空闲时间太长时，将会触发一个 IdleStateEvent 事件。然后，你可以通过在你的 ChannelInboundHandler 中重写 userEventTriggered()方法来处理该 IdleStateEvent 事件
ReadTimeoutHandler	如果在指定的时间间隔内没有收到任何的入站数据，则抛出一个 ReadTimeoutException 并关闭对应的 Channel。可以通过重写你的 ChannelHandler 中的 exceptionCaught()方法来检测该 ReadTimeoutException
WriteTimeoutHandler	如果在指定的时间间隔内没有任何出站数据写入，则抛出一个 WriteTimeoutException 并关闭对应的 Channel。可以通过重写你的 ChannelHandler 的 exceptionCaught()方法检测该 WriteTimeoutException

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(
            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.ISO_8859_1));

        @Override
        public void userEventTriggered(ChannelHandlerContext ctx,
            Object evt) throws Exception {
            if (evt instanceof IdleStateEvent) {
                ctx.writeAndFlush(HEARTBEAT_SEQUENCE.duplicate())
```

1 IdleStateHandler 将在被触发时发送一个 IdleStateEvent 事件

将一个 HeartbeatHandler 添加到 ChannelPipeline 中

实现 userEventTriggered()方法以发送心跳消息

发送到远程节点的心跳消息

2 发送心跳消息，并在发送失败时关闭该连接

```
                    .addListener(
                        ChannelFutureListener.CLOSE_ON_FAILURE);
            } else {
                super.userEventTriggered(ctx, evt);
            }
        }
    }
}
```

不是 IdleStateEvent 事件，所以将它传递给下一个 ChannelInboundHandler

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 IdleStateHandler❶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 ChannelFutureListener❷。

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[①]）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。

表 11-5 用于处理基于分隔符的协议和基于长度的协议的解码器

名 称	描 述
DelimiterBasedFrameDecoder	使用任何由用户提供的分隔符来提取帧的通用解码器
LineBasedFrameDecoder	提取由行尾符（\n 或者\r\n）分隔的帧的解码器。这个解码器比 DelimiterBasedFrameDecoder 更快

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。

① 有关这些协议的 RFC 可以在 IETF 的网站上找到：SMTP 在 www.ietf.org/rfc/rfc2821.txt，POP3 在 www.ietf.org/rfc/rfc1939.txt，IMAP 在 http://tools.ietf.org/html/rfc3501，而 Telnet 在 http://tools.ietf.org/search/rfc854。

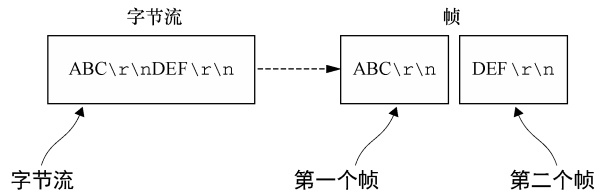


图 11-5 由行尾符分隔的帧

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();
        pipeline.addLast(new LineBasedFrameDecoder(64 * 1024));
        pipeline.addLast(new FrameHandler());
    }

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

该 LineBasedFrameDecoder 将提取的帧转发给下一个 ChannelInboundHandler

添加 FrameHandler 以接收帧

传入了单个帧的内容

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> {
    final byte SPACE = (byte)' ';
    @Override
    protected void initChannel(Channel ch) throws Exception {
        ChannelPipeline pipeline = ch.pipeline();
        pipeline.addLast(new CmdDecoder(64 * 1024));
        pipeline.addLast(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 name() {
            return name;
        }

        public ByteBuf args() {
            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(), SPACE);
            return new Cmd(frame.slice(frame.readerIndex(), index),
                frame.slice(index + 1, frame.writerIndex()));
        }
```

添加 CmdDecoder 以提取 Cmd 对象，并将它转发给下一个 ChannelInboundHandler

添加 CmdHandler 以接收和处理 Cmd 对象

Cmd POJO

从 ByteBuf 中提取由行尾符序列分隔的帧

如果输入中没有帧，则返回 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

使用包含有命令名称和参数的切片创建新的 Cmd 对象

```
    }

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

处理传经 ChannelPipeline 的 Cmd 对象

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。[①]表 11-6 列出了Netty提供的用于处理这种类型的协议的两种解码器。

表 11-6 用于基于长度的协议的解码器

名 称	描 述
FixedLengthFrameDecoder	提取在调用构造函数时指定的定长帧
LengthFieldBasedFrameDecoder	根据编码进帧头部中的长度值提取帧；该字段的偏移量以及长度在构造函数中指定

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。

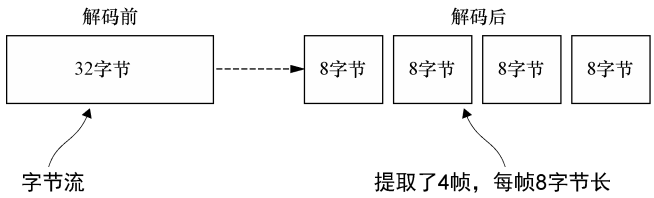


图 11-6 解码长度为 8 字节的帧

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。

① 对于固定帧大小的协议来说，不需要将帧长度编码到头部。——译者注

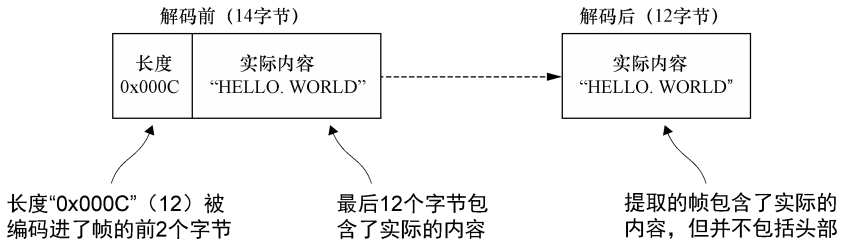


图 11-7 将变长帧大小编码进头部的消息

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

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

```
public class LengthBasedInitializer extends ChannelInitializer<Channel> {
    @Override
    protected void initChannel(Channel ch) throws Exception {
        ChannelPipeline pipeline = ch.pipeline();
        pipeline.addLast(
            new LengthFieldBasedFrameDecoder(64 * 1024, 0, 8));
        pipeline.addLast(new FrameHandler());
    }

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

添加 FrameHandler 以处理每个帧

处理帧的数据

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[①]，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

```
FileInputStream in = new FileInputStream(file);
FileRegion region = new DefaultFileRegion(
    in.getChannel(), 0, file.length());
channel.writeAndFlush(region).addListener(
    new ChannelFutureListener() {
    @Override
    public void operationComplete(ChannelFuture future)
        throws Exception {
        if (!future.isSuccess()) {
            Throwable cause = future.cause();
            // Do something
        }
    }
});
```

创建一个 FileInputStream

以该文件的完整长度创建一个新的 DefaultFileRegion

发送该 DefaultFileRegion，并注册一个 ChannelFutureListener

处理失败

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

① 我们甚至可以利用 io.netty.channel.ChannelProgressivePromise 来实时获取传输的进度。——译者注

用所示的 ChannelHandler 链初始化该 Channel。

表 11-7 **ChunkedInput** 的实现

名 称	描 述
ChunkedFile	从文件中逐块获取数据，当你的平台不支持零拷贝或者你需要转换数据时使用
ChunkedNioFile	和 ChunkedFile 类似，只是它使用了 FileChannel
ChunkedStream	从 InputStream 中逐块传输内容
ChunkedNioStream	从 ReadableByteChannel 中逐块传输内容

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();
        pipeline.addLast(new SslHandler(sslCtx.newEngine(ch.alloc());
        pipeline.addLast(new ChunkedWriteHandler());
        pipeline.addLast(new WriteStreamHandler());
    }

    public final class WriteStreamHandler
        extends ChannelInboundHandlerAdapter {

        @Override
        public void channelActive(ChannelHandlerContext ctx)
            throws Exception {
            super.channelActive(ctx);
            ctx.writeAndFlush(
            new ChunkedStream(new FileInputStream(file)));
        }
    }
}
```

将 SslHandler 添加到 ChannelPipeline 中

添加 ChunkedWriteHandler 以处理作为 ChunkedInput 传入的数据

一旦连接建立，WriteStreamHandler 就开始写文件数据

当连接建立时，channelActive() 方法将使用 ChunkedInput 写文件数据

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[①]。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。

表 11-8 JDK 序列化编解码器

名　称	描　述
`CompatibleObjectDecoder`[②]	和使用 JDK 序列化的非基于 Netty 的远程节点进行互操作的解码器
`CompatibleObjectEncoder`	和使用 JDK 序列化的非基于 Netty 的远程节点进行互操作的编码器
`ObjectDecoder`	构建于 JDK 序列化之上的使用自定义的序列化来解码的解码器；当没有其他的外部依赖时，它提供了速度上的改进。否则其他的序列化实现更加可取
`ObjectEncoder`	构建于 JDK 序列化之上的使用自定义的序列化来编码的编码器；当没有其他的外部依赖时，它提供了速度上的改进。否则其他的序列化实现更加可取

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[③]上的概述中对它是这么定义的：

① 参见 Oracle 的 Java SE 文档中的“JavaObject Serialization”部分：http://docs.oracle.com/javase/8/docs/technotes/guides/serialization/。

② 这个类已经在 Netty 3.1 中废弃，并不存在于 Netty 4.x 中：https://issues.jboss.org/browse/NETTY-136。——译者注

③ “About JBoss Marshalling”：www.jboss.org/jbossmarshalling。

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。

表 11-9 JBoss Marshalling 编解码器

名 称	描 述
CompatibleMarshallingDecoder CompatibleMarshallingEncoder	与只使用 JDK 序列化的远程节点兼容
MarshallingDecoder MarshallingEncoder	适用于使用 JBoss Marshalling 的节点。这些类必须一起使用

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();
        pipeline.addLast(new MarshallingDecoder(unmarshallerProvider));
        pipeline.addLast(new MarshallingEncoder(marshallerProvider));
        pipeline.addLast(new ObjectHandler());
     }

    public static final class ObjectHandler
        extends SimpleChannelInboundHandler<Serializable> {
        @Override
        public void channelRead0(
```

添加 MarshallingDecoder 以将 ByteBuf 转换为 POJO

添加 MarshallingEncoder 以将 POJO 转换为 ByteBuf

添加 ObjectHandler，以处理普通的实现了 Serializable 接口的 POJO

```
            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[①]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。

表 11-10 Protobuf 编解码器

名 称	描 述
`ProtobufDecoder`	使用 protobuf 对消息进行解码
`ProtobufEncoder`	使用 protobuf 对消息进行编码
`ProtobufVarint32FrameDecoder`	根据消息中的 Google Protocol Buffers 的“Base 128 Varints”[a] 整型长度字段值动态地分割所接收到的 `ByteBuf`
`ProtobufVarint32LengthFieldPrepender`	向 `ByteBuf` 前追加一个 Google Protocal Buffers 的“Base 128 Varints”整型的长度字段值

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();
        pipeline.addLast(new ProtobufVarint32FrameDecoder());
```

添加 ProtobufVarint32FrameDecoder 以分隔帧

① 有关 Protocol Buffers 的描述请参考 https://developers.google.com/protocol-buffers/?hl=zh。

```
        pipeline.addLast(new ProtobufEncoder());
        pipeline.addLast(new ProtobufDecoder(lite));
        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
        }
    }
}
```

添加 ProtobufEncoder 以处理消息的编码

添加 ObjectHandler 以处理解码消息

添加 ProtobufDecoder 以解码消息

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。

11.7 小结

Netty 提供的编解码器以及各种 ChannelHandler 可以被组合和扩展，以实现非常广泛的处理方案。此外，它们也是被论证的、健壮的组件，已经被许多的大型系统所使用。

需要注意的是，我们只涵盖了最常见的示例；Netty 的 API 文档提供了更加全面的覆盖。

在下一章中，我们将学习另一种先进的协议——WebSocket，它被开发用以改进 Web 应用程序的性能以及响应性。Netty 提供了你将会需要的工具，以便你快速、轻松地利用它强大的功能。

① 还需要在当前的 ProtobufEncoder 之前添加一个相应的 ProtobufVarint32LengthFieldPrepender 以编码进帧长度信息。——译者注