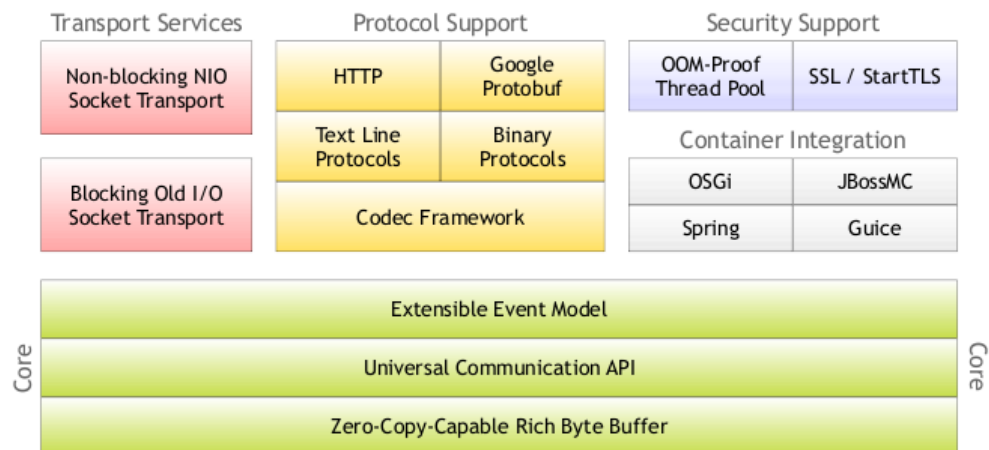


Netty 3.1 中文用户手册



下载: <http://www.codepub.com/software/view-software-17736.html>

在线阅读: <http://edu.codepub.com/2010/0413/21990.php>

The Netty Project 3.1 User Guide

The Proven Approach to Rapid Network Application Development

目录

序言	3
1. 问题	3
2. 方案	3
第一章. 开始	3
1.1. 开始之前	4
1.2. 抛弃协议服务	4
1.3. 查看接收到的数据	8
1.4. 响应协议服务	9
1.5. 时间协议服务	9
1.6. 时间协议服务客户端	12
1.7. 流数据的传输处理	15
1.8. 使用 POJO 代替 ChannelBuffer	22
1.9. 关闭你的应用	26
1.10. 总述	30
第二章. 架构总览	30
2.1. 丰富的缓冲实现	31
2.2. 统一的异步 I/O API	31
2.3. 基于拦截链模式的事件模型	32
2.4. 适用快速开发的高级组件	33
2.5. 总述	34

序言

本指南对 [Netty](#) 进行了介绍并指出其意义所在。

1. 问题

现在，我们使用适合一般用途的应用或组件来和彼此通信。例如，我们常常使用一个 HTTP 客户端从远程服务器获取信息或者通过 **web services** 进行远程方法的调用。

然而，一个适合普通目的的协议或其实现并不具备其规模上的扩展性。例如，我们无法使用一个普通的 HTTP 服务器进行大型文件，电邮信息的交互，或者处理金融信息和多人游戏数据那种要求准实时消息传递的应用场景。因此，这些都要求使用一个适用于特殊目的并经过高度优化的协议实现。例如，你可能想要实现一个对基于 **AJAX** 的聊天应用，媒体流或大文件传输进行过特殊优化的 HTTP 服务器。你甚至可能想去设计和实现一个全新的，特定于你的需求的通信协议。

另一种无法避免的场景是你可能不得不使用一种专有的协议和原有系统交互。在这种情况下，你需要考虑的是如何能够快速开发出这个协议的实现并且同时还没有牺牲最终应用的性能和稳定性。

2. 方案

[Netty](#) 是一个异步的，事件驱动的网络编程框架和工具，使用 [Netty](#) 可以快速开发出可维护的，高性能、高扩展能力的协议服务及其客户端应用。

也就是说，[Netty](#) 是一个基于 **NIO** 的客户，服务器端编程框架，使用 [Netty](#) 可以确保你快速和简单的开发出一个网络应用，例如实现了某种协议的客户端，服务端应用。[Netty](#) 相当简化和流线化了网络应用的编程开发过程，例如，**TCP** 和 **UDP** 的 **socket** 服务开发。

“快速”和“简单”并不意味着会让你的最终应用产生维护性或性能上的问题。[Netty](#) 是一个吸收了多种协议的实现经验，这些协议包括 **FTP**,**SMTP**,**HTTP**，各种二进制，文本协议，并经过相当精心设计的项目，最终，[Netty](#) 成功的找到了一种方式，在保证易于开发的同时还保证了其应用的性能，稳定性和伸缩性。

一些用户可能找到了某些同样声称具有这些特性的编程框架，因此你们可能想问 [Netty](#) 又有什么不一样的地方。这个问题的答案是 [Netty](#) 项目的设计哲学。从创立之初，无论是在 **API** 还是在其实现上 [Netty](#) 都致力于为你提供最为舒适的使用体验。虽然这并不是显而易见的，但你终将会认识到这种设计哲学将令你在阅读本指南和使用 [Netty](#) 时变得更加得轻松和容易。

第一章. 开始

这一章节将围绕 Netty 的核心结构展开，同时通过一些简单的例子可以让你更快的了解 Netty 的使用。当你读完本章，你将有能力使用 Netty 完成客户端和服务端的开发。

如果你更喜欢自上而下式的学习方式，你可以首先完成 第二章：架构总览 的学习，然后再回到这里。

1.1. 开始之前

运行本章示例程序的两个最低要求是：最新版本的 Netty 程序以及 JDK 1.5或更高版本。最新版本的 Netty 程序可在[项目下载页](#) 下载。下载正确版本的 JDK，请到你偏好的 JDK 站点下载。


这已经足够了吗？实际上你会发现，这两个条件已经足够你完成任何协议的开发了。如果不是这样，请[联系 Netty 项目社区](#)，让我们知道还缺少了什么。

最终但不是至少，当你想了解本章所介绍的类的更多信息时请参考 API 手册。为方便你的使用，这篇文档中所有的类名均连接至在线 API 手册。此外，如果本篇文档中有任何错误信息，无论是语法错误，还是打印排版错误或者你有更好的建议，请不要顾虑，立即[联系 Netty 项目社区](#)。

1.2. 抛弃协议服务

在这个世界上最简化的协议不是“Hello,world!”而是[抛弃协议](#)。这是一种丢弃接收到的任何数据并不做任何回应的协议。

实现抛弃协议(DISCARD protocol)，你仅需要忽略接受到的任何数据即可。让我们直接从处理器(handler)实现开始，这个处理器处理 Netty 的所有 I/O 事件。

Java 代码 

```
1 package org.jboss.netty.example.discard;
2
3 @ChannelPipelineCoverage("all")1
4 public class DiscardServerHandler extends SimpleChannelHandler {2
5
6     @Override
7     public void messageReceived(ChannelHandlerContext ctx, MessageEvent e) {3
```

```

8      }

9

10     @Override

11     public void exceptionCaught(ChannelHandlerContext ctx, ExceptionEvent e) {4

12         e.getCause().printStackTrace();

13

14         Channel ch = e.getChannel();

15         ch.close();

16     }


17 }

```

代码说明

- 1) `ChannelPipelineCoverage` 注解了一种处理器类型，这个注解标示了一个处理器是否可被多个 `Channel` 通道共享（同时关联着 `ChannelPipeline`）。 `DiscardServerHandler` 没有处理任何有状态的信息，因此这里的注解是“all”。
- 2) `DiscardServerHandler` 继承了 `SimpleChannelHandler`，这也是一个 `ChannelHandler` 的实现。`SimpleChannelHandler` 提供了多种你可以重写的事件处理方法。目前直接继承 `SimpleChannelHandler` 已经足够了，并不需要你完成一个自己的处理器接口。
- 3) 我们这里重写了 `messageReceived` 事件处理方法。这个方法由一个接收了客户端传送数据的 `MessageEvent` 事件调用。在这个例子中，我们忽略接收到的任何数据，并以此来实现一个抛弃协议（DISCARD protocol）。
- 4) `exceptionCaught` 事件处理方法由一个 `ExceptionEvent` 异常事件调用，这个异常事件起因于 `Netty` 的 I/O 异常或一个处理器实现的内部异常。多数情况下，捕捉到的异常应当被记录下来，并在这个方法中关闭这个 `channel` 通道。当然处理这种异常情况的方法实现可能因你的实际需求而有所不同，例如，在关闭这个连接之前你可能会发送一个包含了错误码的响应消息。

目前进展不错，我们已经完成了抛弃协议服务器的一半开发工作。下面要做的是完成一个可以启动这个包含 `DiscardServerHandler` 处理器服务的主方法。

Java 代码 

```
18 package org.jboss.netty.example.discard;

19

20 import java.net.InetSocketAddress;

21 import java.util.concurrent.Executors;

22

23 public class DiscardServer {

24

25     public static void main(String[] args) throws Exception {

26         ChannelFactory factory =

27             new NioServerSocketChannelFactory (

28                 Executors.newCachedThreadPool(),

29                 Executors.newCachedThreadPool());

30

31         ServerBootstrap bootstrap = new ServerBootstrap (factory);

32

33         DiscardServerHandler handler = new DiscardServerHandler();

34         ChannelPipeline pipeline = bootstrap.getPipeline();

35         pipeline.addLast("handler", handler);

36
```

```
37         bootstrap.setOption("child.tcpNoDelay", true);

38         bootstrap.setOption("child.keepAlive", true);

39

40         bootstrap.bind(new InetSocketAddress(8080));

41     }

42 }
```

代码说明

1) **ChannelFactory** 是一个创建和管理 **Channel** 通道及其相关资源的工厂接口，它处理所有的 I/O 请求并产生相应的 I/O **ChannelEvent** 通道事件。**Netty** 提供了多种 **ChannelFactory** 实现。这里我们需要实现一个服务端的例子，因此我们使用 **NioServerSocketChannelFactory** 实现。另一件需要注意的事情是这个工厂并不自己负责创建 I/O 线程。你应当在其构造器中指定该工厂使用的线程池，这样做的好处是你获得了更高的控制力来管理你的应用环境中使用的线程，例如一个包含了安全管理的应用服务。

2) **ServerBootstrap** 是一个设置服务的帮助类。你甚至可以在这个服务中直接设置一个 **Channel** 通道。然而请注意，这是一个繁琐的过程，大多数情况下并不需要这样做。

3) 这里，我们将 **DiscardServerHandler** 处理器添加至默认的 **ChannelPipeline** 通道。任何时候当服务器接收到一个新的连接，一个新的 **ChannelPipeline** 管道对象将被创建，并且所有在这里添加的 **ChannelHandler** 对象将被添加至这个新的 **ChannelPipeline** 管道对象。这很像是一种浅拷贝操作（a shallow-copy operation）；所有的 **Channel** 通道以及其对应的 **ChannelPipeline** 实例将分享相同的 **DiscardServerHandler** 实例。

4) 你也可以设置我们在这里指定的这个通道实现的配置参数。我们正在写的是一个 TCP/IP 服务，因此我们运行设定一些 **socket** 选项，例如 **tcpNoDelay** 和 **keepAlive**。请注意我们在配置选项里添加的 "child." 前缀。这意味着这个配置项仅适用于我们接收到的通道实例，而不是 **ServerSocketChannel** 实例。因此，你可以这样给一个 **ServerSocketChannel** 设定参数：

```
bootstrap.setOption("reuseAddress", true);
```

5) 我们继续。剩下要做的是绑定这个服务使用的端口并且启动这个服务。这里，我们绑定本机所有网卡（NICs, network interface cards）上的 8080 端口。当然，你现在也可以对应不同的绑定地址多次调用绑定操作。


大功告成！现在你已经完成你的第一个基于 **Netty** 的服务端程序。

1.3. 查看接收到的数据

现在你已经完成了你的第一个服务端程序，我们需要测试它是否可以真正的工作。最简单的方法是使用 **telnet** 命令。例如，你可以在命令行中输入“**telnet localhost 8080**”或其他类型参数。

然而，我们可以认为服务器在正常工作吗？由于这是一个丢球协议服务，所以实际上我们无法真正的知道。你最终将收不到任何回应。为了证明它在真正的工作，让我们修改代码打印其接收到的数据。

我们已经知道当完成数据的接收后将产生 **MessageEvent** 消息事件，并且也会触发 **messageReceived** 处理方法。所以让我在 **DiscardServerHandler** 处理器的 **messageReceived** 方法内增加一些代码。

Java 代码 

```
43 @Override
44 public void messageReceived(ChannelHandlerContext ctx, MessageEvent e) {
45     ChannelBuffer buf = (ChannelBuffer) e.getMessage();
46     while(buf.readable()) {
47         System.out.println((char) buf.readByte());
48     }
49 }
```

代码说明

1) 基本上我们可以假定在 **socket** 的传输中消息类型总是 **ChannelBuffer**。**ChannelBuffer** 是 **Netty** 的一个基本数据结构，这个数据结构存储了一个字节序列。**ChannelBuffer** 类似于 **NIO** 的 **ByteBuffer**，但是前者却更加的灵活和易于使用。例如，**Netty** 允许你创建一个由多个 **ChannelBuffer** 构建的复合 **ChannelBuffer** 类型，这样就可以减少不必要的内存拷贝次数。

2) 虽然 **ChannelBuffer** 有些类似于 **NIO** 的 **ByteBuffer**，但强烈建议你参考 **Netty** 的 API 手册。学会如何正确的使用 **ChannelBuffer** 是无障碍使用 **Netty** 的关键一步。


如果你再次运行 **telnet** 命令，你将会看到你所接收到的数据。

抛弃协议服务的所有源代码均存放在在分发版的 **org.jboss.netty.example.discard** 包下。

1.4. 响应协议服务

目前，我们虽然使用了数据，但最终却未作任何回应。然而一般情况下，一个服务都需要回应一个请求。让我们实现 [ECHO 协议](#) 来学习如何完成一个客户请求的回应消息，ECHO 协议规定要返回任何接收到的数据。

与我们上一节实现的抛弃协议服务唯一不同的地方是，这里需要返回所有的接收数据而不是仅仅打印在控制台之上。因此我们再次修改 `messageReceived` 方法就足够了。

Java 代码 

```
50 @Override

51 public void messageReceived(ChannelHandlerContext ctx, MessageEvent e) {

52     Channel ch = e.getChannel();

53     ch.write(e.getMessage());

54 }
```

代码说明

1) 一个 `ChannelEvent` 通道事件对象自身存有一个和其关联的 `Channel` 对象引用。这个返回的 `Channel` 通道对象代表了这个接收 `MessageEvent` 消息事件的连接（connection）。因此，我们可以通过调用这个 `Channel` 通道对象的 `write` 方法向远程节点写入返回数据。


现在如果你再次运行 `telnet` 命令，你将会看到服务器返回的你所发送的任何数据。

相应服务的所有源代码存放在分发版的 `org.jboss.netty.example.echo` 包下。

1.5. 时间协议服务

这一节需要实现的协议是 [TIME 协议](#)。这是一个与先前所介绍的不同的例子。这个例子里，服务端返回一个32位的整数消息，我们不接受请求中包含的任何数据并且当消息返回完毕后立即关闭连接。通过这个例子你将学会如何构建和发送消息，以及当完成处理后如何主动关闭连接。

因为我们会忽略接收到的任何数据而只是返回消息，这应当在建立连接后就立即开始。因此这次我们不再使用 `messageReceived` 方法，取而代之的是使用 `channelConnected` 方法。下面是具体的实现：

Java 代码 

```
55 package org.jboss.netty.example.time;

56

57 @ChannelPipelineCoverage("all")

58 public class TimeServerHandler extends SimpleChannelHandler {

59

60     @Override

61     public void channelConnected(ChannelHandlerContext ctx, ChannelStateEvent e)

62     {

63         Channel ch = e.getChannel();

64

65         ChannelBuffer time = ChannelBuffers.buffer(4);

66         time.writeInt(System.currentTimeMillis() / 1000);

67

68         ChannelFuture f = ch.write(time);

69         f.addListener(new ChannelFutureListener() {

70             public void operationComplete(ChannelFuture future) {

71                 Channel ch = future.getChannel();

72                 ch.close();

73             }

74         });

75     }

76 }
```

```

74         });

75     }

76

77     @Override

78     public void exceptionCaught(ChannelHandlerContext ctx, ExceptionEvent e) {

79         e.getCause().printStackTrace();

80         e.getChannel().close();

81     }

82 }

```

代码说明

1) 正如我们解释过的，`channelConnected` 方法将在一个连接建立后立即触发。因此让我们在这个方法里完成一个代表当前时间（秒）的32位整数消息的构建工作。

2) 为了发送一个消息，我们需要分配一个包含了这个消息的 `buffer` 缓冲。因为我们将要写入一个32位的整数，因此我们需要一个4字节的 `ChannelBuffer`。`ChannelBuffers` 是一个可以创建 `buffer` 缓冲的帮助类。除了这个 `buffer` 方法，`ChannelBuffers` 还提供了很多和 `ChannelBuffer` 相关的实用方法。更多信息请参考 API 手册。

另外，一个很不错的方法是使用静态的导入方式：

```

import static org.jboss.netty.buffer.ChannelBuffers.*;

...

ChannelBuffer dynamicBuf = dynamicBuffer(256);
ChannelBuffer ordinaryBuf = buffer(1024);

```

3) 像通常一样，我们需要自己构造消息。

但是打住，`flip` 在哪？过去我们在使用 NIO 发送消息时不是常常需要调用 `ByteBuffer.flip()` 方法吗？实际上 `ChannelBuffer` 之所以不需要这个方法是因为 `ChannelBuffer` 有两个指针：一个对应读操作，一个对应写操作。当你向一个 `ChannelBuffer` 写入数据的时候写指针的索引值便会增加，但与此同时读

指针的索引值不会有任何变化。读写指针的索引值分别代表了这个消息的开始、结束位置。

与之相应的是，NIO 的 **buffer** 缓冲没有为我们提供如此简洁的一种方法，除非你调用它的 **flip** 方法。因此，当你忘记调用 **flip** 方法而引起发送错误时，你便会陷入困境。这样的错误不会再 **Netty** 中发生，因为我们对应不同的操作类型有不同的指针。你会发现就像你已习惯的这样过程变得更加容易——一种没有 **flipping** 的体验！

另一点需要注意的是这个写方法返回了一个 **ChannelFuture** 对象。一个 **ChannelFuture** 对象代表了一个尚未发生的 I/O 操作。这意味着，任何已请求的操作都可能是没有被立即执行的，因为在 **Netty** 内部所有的操作都是异步的。例如，下面的代码可能会关闭一个连接，这个操作甚至会发生消息发送之前：

```
Channel ch = ...;
ch.write(message);
ch.close();
```

因此，你需要这个 **write** 方法返回的 **ChannelFuture** 对象，**close** 方法需要等待写操作异步完成之后的 **ChannelFuture** 通知/监听触发。需要注意的是，关闭方法仍旧不是立即关闭一个连接，它同样也是返回了一个 **ChannelFuture** 对象。

4) 在写操作完成之后我们又如何得到通知？这个只需要简单的为这个返回的 **ChannelFuture** 对象增加一个 **ChannelFutureListener** 即可。在这里我们创建了一个匿名 **ChannelFutureListener** 对象，在这个 **ChannelFutureListener** 对象内部我们处理了异步操作完成之后的关闭操作。


另外，你也可以通过使用一个预定义的监听类来简化代码。

```
f.addListener(ChannelFutureListener.CLOSE);
```

1.6. 时间协议服务客户端

不同于 **DISCARD** 和 **ECHO** 协议服务，我们需要一个时间协议服务的客户端，因为人们无法直接将一个32位的二进制数据转换一个日历时间。在这一节我们将学习如何确保服务器端工作正常，以及如何使用 **Netty** 完成客户端的开发。

使用 **Netty** 开发服务器端和客户端代码最大的不同是要求使用不同的 **Bootstrap** 及 **ChannelFactory**。请参照以下的代码：

Java 代码 

```
83 package org.jboss.netty.example.time;

84

85 import java.net.InetSocketAddress;

86 import java.util.concurrent.Executors;

87

88 public class TimeClient {

89

90     public static void main(String[] args) throws Exception {

91         String host = args[0];

92         int port = Integer.parseInt(args[1]);

93

94         ChannelFactory factory =

95             new NioClientSocketChannelFactory (

96                 Executors.newCachedThreadPool(),

97                 Executors.newCachedThreadPool());

98

99         ClientBootstrap bootstrap = new ClientBootstrap (factory);

100

101         TimeClientHandler handler = new TimeClientHandler();

102         bootstrap.getPipeline().addLast("handler", handler);
```

```
103
104         bootstrap.setOption("tcpNoDelay" , true);
105         bootstrap.setOption("keepAlive", true);
106
107         bootstrap.connect (new InetSocketAddress(host, port));
108     }
109 }
```

代码说明

- 1) 使用 `NioClientSocketChannelFactory` 而不是 `NioServerSocketChannelFactory` 来创建客户端的 `Channel` 通道对象。
- 2) 客户端的 `ClientBootstrap` 对应 `ServerBootstrap`。
- 3) 请注意，这里不存在使用“child.”前缀的配置项，客户端的 `SocketChannel` 实例不存在父级 `Channel` 对象。
- 4) 我们应当调用 `connect` 连接方法，而不是之前的 `bind` 绑定方法。

正如你所看到的，这与服务端的启动过程是完全不一样的。`ChannelHandler` 又该如何实现呢？它应当负责接收一个32位的整数，将其转换为可读的格式后，打印输出时间，并关闭这个连接。

Java 代码

```
110 package org.jboss.netty.example.time;
111
112 import java.util.Date;
113
114 @ChannelPipelineCoverage("all")
```

```

115 public class TimeClientHandler extends SimpleChannelHandler {
116
117     @Override
118     public void messageReceived(ChannelHandlerContext ctx, MessageEvent e) {
119         ChannelBuffer buf = (ChannelBuffer) e.getMessage();
120         long currentTimeMillis = buf.readInt() * 1000L;
121         System.out.println(new Date(currentTimeMillis));
122         e.getChannel().close();
123     }
124
125     @Override
126     public void exceptionCaught(ChannelHandlerContext ctx, ExceptionEvent e) {
127         e.getCause().printStackTrace();
128         e.getChannel().close();
129     }
130 }

```

这看起来很简单，与服务端的实现也并未有什么不同。然而，这个处理器却时常会因为抛出 `IndexOutOfBoundsException` 异常而拒绝工作。我们将在下一节讨论这个问题产生的原因。

1.7. 流数据的传输处理

1.7.1. Socket Buffer 的缺陷

对于例如 **TCP/IP** 这种基于流的传输协议实现，接收到的数据会被存储在 **socket** 的接受缓冲区内。不幸的是，这种基于流的传输缓冲区并不是一个包队列，而是一个字节队列。这意味着，即使你以两个数据包的形式发送了两条消息，操作系统却不会把它们看成是两条消息，而仅仅是一个批次的字节序列。因此，在这种情况下我们就无法保证收到的数据恰好就是远程节点所发送的数据。例如，让我们假设一个操作系统的 **TCP/IP** 堆栈收到了三个数据包：

```
+-----+-----+-----+
| ABC | DEF | GHI |
+-----+-----+-----+
```

由于这种流传输协议的普遍性质，在你的应用中有较高的可能会把这些数据读取为另外一种形式：

```
+-----+-----+---+---+
| AB | CDEFG | H | I |
+-----+-----+---+---+
```


因此对于数据的接收方，不管是服务端还是客户端，应当重构这些接收到的数据，让其变成一种可让你的应用逻辑易于理解的更有意义的数据结构。在上面所述的这个例子中，接收到的数据应当重构为下面的形式：

```
+-----+-----+-----+
| ABC | DEF | GHI |
+-----+-----+-----+
```

1.7.2. 第一种方案

现在让我们回到时间协议服务客户端的例子中。我们在这里遇到了同样的问题。一个**32位**的整数是一个非常小的数据量，因此它常常不会被切分在不同的数据段内。然而，问题是它确实可以被切分在不同的数据段内，并且这种可能性随着流量的增加而提高。

最简单的方案是在程序内部创建一个可准确接收**4**字节数据的累积性缓冲。下面的代码是修复了这个问题后的 **TimeClientHandler** 实现。

Java 代码 

```
131 package org.jboss.netty.example.time;
132
```



```
133 import static org.jboss.netty.buffer.ChannelBuffers.*;

134

135 import java.util.Date;

136

137 @ChannelPipelineCoverage("one")

138 public class TimeClientHandler extends SimpleChannelHandler {

139

140     private final ChannelBuffer buf = dynamicBuffer();

141

142     @Override

143     public void messageReceived(ChannelHandlerContext ctx, MessageEvent e) {

144         ChannelBuffer m = (ChannelBuffer) e.getMessage();

145         buf.writeBytes(m);

146

147         if (buf.readableBytes() >= 4) {

148             long currentTimeMillis = buf.readInt() * 1000L;

149             System.out.println(new Date(currentTimeMillis));

150             e.getChannel().close();

151         }

152     }
```

```


153
154     @Override
155     public void exceptionCaught(ChannelHandlerContext ctx, ExceptionEvent e) {
156         e.getCause().printStackTrace();
157         e.getChannel().close();
158     }
159 }

```

代码说明

- 1) 这一次我们使用“one”做为 `ChannelPipelineCoverage` 的注解值。这是由于这个修改后的 `TimeClientHandler` 不在不在内部保持一个 `buffer` 缓冲，因此这个 `TimeClientHandler` 实例不可以再被多个 `Channel` 通道或 `ChannelPipeline` 共享。否则这个内部的 `buffer` 缓冲将无法缓冲正确的数据内容。
- 2) 动态的 `buffer` 缓冲也是 `ChannelBuffer` 的一种实现，其拥有动态增加缓冲容量的能力。当你无法预估消息的数据长度时，动态的 `buffer` 缓冲是一种很有用的缓冲结构。
- 3) 首先，所有的数据将会被累积的缓冲至 `buf` 容器。
- 4) 之后，这个处理器将会检查是否收到了足够的数据然后再进行真实的业务逻辑处理，在这个例子中需要接收4字节数据。否则，`Netty` 将重复调用 `messageReceived` 方法，直至4字节数据接收完成。

这里还有另一个地方需要进行修改。你是否还记得我们把 `TimeClientHandler` 实例添加到了这个 `ClientBootstrap` 实例的默认 `ChannelPipeline` 管道里？这意味着同一个 `TimeClientHandler` 实例将被多个 `Channel` 通道共享，因此接受的数据也将受到破坏。为了给每一个 `Channel` 通道创建一个新的 `TimeClientHandler` 实例，我们需要实现一个 `ChannelPipelineFactory` 管道工厂：

Java 代码 


```

160 package org.jboss.netty.example.time;
161
162 public class TimeClientPipelineFactory implements ChannelPipelineFactory {

```


```
163
164     public ChannelPipeline getPipeline() {
165         ChannelPipeline pipeline = Channels.pipeline();
166         pipeline.addLast("handler", new TimeClientHandler());
167         return pipeline;
168     }
169 }
```

现在，我们需要把 `TimeClient` 下面的代码片段：

Java 代码 

```
170 TimeClientHandler handler = new TimeClientHandler();
171 bootstrap.getPipeline().addLast("handler", handler);
```

替换为：

Java 代码 

```
172 bootstrap.setPipelineFactory(new TimeClientPipelineFactory());
```

虽然这看上去有些复杂，并且由于在 `TimeClient` 内部我们只创建了一个连接（`connection`），因此我们在这里确实没必要引入 `TimeClientPipelineFactory` 实例。

然而，当你的应用变得越来越复杂，你就总会需要实现自己的 `ChannelPipelineFactory`，这个管道工厂将会令你的管道配置变得更加具有灵活性。

1.7.3. 第二种方案


虽然第二种方案解决了时间协议客户端遇到的问题，但是这个修改后的处理器实现看上去却不再那么简洁。设想一种更为复杂的，由多个可变长度字段组成的协议。你的 `ChannelHandler` 实现将变得越来越难以维护。

正如你已注意到的，你可以为一个 `ChannelPipeline` 添加多个 `ChannelHandler`，因此，为了减小应用的复杂性，你可以把这个臃肿的 `ChannelHandler` 切分为多个独立的模块单元。例如，你可以把

TimeClientHandler 切分为两个独立的处理器：

- TimeDecoder，解决数据分段的问题。
- TimeClientHandler，原始版本的实现。

幸运的是，Netty 提供了一个可扩展的类，这个类可以直接拿过来使用帮你完成 TimeDecoder 的开发：

Java 代码 

```
173 package org.jboss.netty.example.time;

174

175

176 public class TimeDecoder extends FrameDecoder {

177

178     @Override

179     protected Object decode(

180         ChannelHandlerContext ctx, Channel channel, ChannelBuffer buffer) {

181

182         if (buffer.readableBytes() < 4) {

183             return null;

184         }

185

186         return buffer.readBytes(4);

187     }

188 }
```

代码说明


1) 这里不再需要使用 ChannelPipelineCoverage 的注解，因为 FrameDecoder 总是被注解为“one”。

2) 当接收到新的数据后，FrameDecoder 会调用 decode 方法，同时传入一个 FrameDecoder 内部持有的累积型 buffer 缓冲。

3) 如果 decode 返回 null 值，这意味着还没有接收到足够的数据。当有足够数量的数据后 FrameDecoder 会再次调用 decode 方法。

4) 如果 decode 方法返回一个非空值，这意味着 decode 方法已经成功完成一条信息的解码。FrameDecoder 将丢弃这个内部的累计型缓冲。请注意你不需要对多条消息进行解码，FrameDecoder 将保持对 decode 方法的调用，直到 decode 方法返回非空对象。

如果你是一个勇于尝试的人，你或许应当使用 ReplayingDecoder，ReplayingDecoder 更加简化解码的过程。为此你需要查看 API 手册获得更多的帮助信息。

Java 代码 

```
189 package org.jboss.netty.example.time;
190
191 public class TimeDecoder extends ReplayingDecoder<VoidEnum> {
192
193     @Override
194     protected Object decode(
195         ChannelHandlerContext ctx, Channel channel,
196         ChannelBuffer buffer, VoidEnum state) {
197
198         return buffer.readBytes(4);
199     }
200 }
```

此外，Netty 还为你提供了一些可以直接使用的 `decoder` 实现，这些 `decoder` 实现不仅可以让你非常容易的实现大多数协议，并且还会帮你避免某些臃肿、难以维护的处理器实现。请参考下面的代码包获得更加详细的实例：


- `org.jboss.netty.example.factorial` for a binary protocol, and
- `org.jboss.netty.example.telnet` for a text line-based protocol

1.8. 使用 POJO 代替 ChannelBuffer

目前为止所有的实例程序都是使用 `ChannelBuffer` 做为协议消息的原始数据结构。在这一节，我们将改进时间协议服务的客户/服务端实现，使用 [POJO](#) 而不是 `ChannelBuffer` 做为协议消息的原始数据结构。

在你的 `ChannelHandler` 实现中使用 POJO 的优势是很明显的；从你的 `ChannelHandler` 实现中分离从 `ChannelBuffer` 获取数据的代码，将有助于提高你的 `ChannelHandler` 实现的可维护性和可重用性。在时间协议服务的客户/服务端代码中，直接使用 `ChannelBuffer` 读取一个32位的整数并不是一个主要的问题。然而，你会发现，当你试图实现一个真实的协议的时候，这种代码上的分离是很有必要的。

首先，让我们定义一个称之为 `UnixTime` 的新类型。

Java 代码 


```
201 package org.jboss.netty.example.time;
202
203 import java.util.Date;
204
205 public class UnixTime {
206     private final int value;
207
208     public UnixTime(int value) {
209         this.value = value;
210     }
211
```

```

212     public int getValue() {
213         return value;
214     }
215
216     @Override
217     public String toString() {
218         return new Date(value * 1000L).toString();
219     }
220 }

```

现在让我们重新修改 `TimeDecoder` 实现，让其返回一个 `UnixTime`，而不是一个 `ChannelBuffer`。

Java 代码 

```


221 @Override
222 protected Object decode(
223     ChannelHandlerContext ctx, Channel channel, ChannelBuffer buffer) {
224     if (buffer.readableBytes() < 4) {
225         return null;
226     }
227
228     return new UnixTime(buffer.readInt());
229 }

```

`FrameDecoder` 和 `ReplayingDecoder` 允许你返回一个任何类型的对象。如果它们仅允许返回一个

ChannelBuffer 类型的对象，我们将不得不插入另一个可以从 ChannelBuffer 对象转换 为 UnixTime 对象的 ChannelHandler 实现。

有了这个修改后的 decoder 实现，这个 TimeClientHandler 便不会再依赖 ChannelBuffer。

Java 代码 

```
230 @Override

231 public void messageReceived(ChannelHandlerContext ctx, MessageEvent e) {


232     UnixTime m = (UnixTime) e.getMessage();

233     System.out.println(m);

234     e.getChannel().close();

235 }
```

更加简单优雅了，不是吗？同样的技巧也可以应用在服务端，让我们现在更新 TimeServerHandler 的实现：

Java 代码 

```
236 @Override

237 public void channelConnected(ChannelHandlerContext ctx, ChannelStateEvent e) {


238     UnixTime time = new UnixTime(System.currentTimeMillis() / 1000);

239     ChannelFuture f = e.getChannel().write(time);

240     f.addListener(ChannelFutureListener.CLOSE);

241 }
```

现在剩下的唯一需要修改的部分是这个 ChannelHandler 实现，这个 ChannelHandler 实现需要把一个 UnixTime 对象重新转换为一个 ChannelBuffer。但这却已是相当简单了，因为当你对消息进行编码的时候你不再需要处理数据包的拆分及组装。

Java 代码 

```
242 package org.jboss.netty.example.time;

243
```



```
244 import static org.jboss.netty.buffer.ChannelBuffers.*;

245

246 @ChannelPipelineCoverage("all")

247 public class TimeEncoder extends SimpleChannelHandler {

248

249     public void writeRequested(ChannelHandlerContext ctx, MessageEvent e) {

250         UnixTime time = (UnixTime) e.getMessage();

251

252         ChannelBuffer buf = buffer(4);

253         buf.writeInt(time.getValue());

254

255         Channels.write(ctx, e.getFuture(), buf);

256     }

257 }
```

代码说明

- 1) 因为这个 `encoder` 是无状态的，所以其使用的 `ChannelPipelineCoverage` 注解值是“all”。实际上，大多数 `encoder` 实现都是无状态的。
- 2) 一个 `encoder` 通过重写 `writeRequested` 方法来实现对写操作请求的拦截。不过请注意虽然这个 `writeRequested` 方法使用了和 `messageReceived` 方法一样的 `MessageEvent` 参数，但是它们却分别对应了不同的解释。一个 `ChannelEvent` 事件可以既是一个上升流事件（`upstream event`）也可以是一个下降流事件（`downstream event`），这取决于事件流的方向。例如：一个 `MessageEvent` 消息事件可以作为一个上升流事件（`upstream event`）被 `messageReceived` 方法调用，也可以作为一个下降流事件（`downstream event`）被 `writeRequested` 方法调用。请参考 API 手册获得上升流事件（`upstream event`）和下降流事件（`downstream event`）的更多信息。

3) 一旦完成了 POJO 和 `ChannelBuffer` 转换，你应当确保把这个新的 `buffer` 缓冲转发至先前的 `ChannelDownstreamHandler` 处理，这个下降通道的处理器由某个 `ChannelPipeline` 管理。`Channels` 提供了多个可以创建和发送 `ChannelEvent` 事件的帮助方法。在这个例子中，`Channels.write(...)` 方法创建了一个新的 `MessageEvent` 事件，并把这个事件发送给了先前的处于某个 `ChannelPipeline` 内的 `ChannelDownstreamHandler` 处理器。

另外，一个很不错的方法是使用静态的方式导入 `Channels` 类：

```
import static org.jboss.netty.channel.Channels.*;

...

ChannelPipeline pipeline = pipeline();

write(ctx, e.getFuture(), buf);

fireChannelDisconnected(ctx);
```

最后的任务是把这个 `TimeEncoder` 插入服务端的 `ChannelPipeline`，这是一个很简单的步骤。


1.9. 关闭你的应用

如果你运行了 `TimeClient`，你肯定可以注意到，这个应用并没有自动退出而只是在那里保持着无意义的运行。跟踪堆栈记录你可以发现，这里有一些运行状态的 I/O 线程。为了关闭这些 I/O 线程并让应用优雅的退出，你需要释放这些由 `ChannelFactory` 分配的资源。

一个典型的网络应用的关闭过程由以下三步组成：

- 关闭负责接收所有请求的 `server socket`。
- 关闭所有客户端 `socket` 或服务端为响应某个请求而创建的 `socket`。
- 释放 `ChannelFactory` 使用的所有资源。

为了让 `TimeClient` 执行这三步，你需要在 `TimeClient.main()` 方法内关闭唯一的客户连接以及 `ChannelFactory` 使用的所有资源，这样做便可以优雅的关闭这个应用。

Java 代码 

```
258 package org.jboss.netty.example.time;

259

260 public class TimeClient {
```

```

261     public static void main(String[] args) throws Exception {
262         ...
263         ChannelFactory factory = ...;
264         ClientBootstrap bootstrap = ...;
265         ...
266         ChannelFuture future = bootstrap.connect(...);
267         future.awaitUninterruptible();
268         if (!future.isSuccess()) {
269             future.getCause().printStackTrace();
270         }
271         future.getChannel().getCloseFuture().awaitUninterruptibly();
272         factory.releaseExternalResources();
273     }
274 }

```

代码说明

- 1) `ClientBootstrap` 对象的 `connect` 方法返回一个 `ChannelFuture` 对象，这个 `ChannelFuture` 对象将告知这个连接操作的成功或失败状态。同时这个 `ChannelFuture` 对象也保存了一个代表这个连接操作的 `Channel` 对象引用。
- 2) 阻塞式的等待，直到 `ChannelFuture` 对象返回这个连接操作的成功或失败状态。
- 3) 如果连接失败，我们将打印连接失败的原因。如果连接操作没有成功或者被取消，`ChannelFuture` 对象的 `getCause()` 方法将返回连接失败的原因。
- 4) 现在，连接操作结束，我们需要等待并且一直到这个 `Channel` 通道返回的 `closeFuture` 关闭这个连接。每一个 `Channel` 都可获得自己的 `closeFuture` 对象，因此我们可以收到通知并在这个关闭时间点执

行某种操作。


并且即使这个连接操作失败，这个 `closeFuture` 仍旧会收到通知，因为这个代表连接的 `Channel` 对象将会在连接操作失败后自动关闭。

5) 在这个时间点，所有的连接已被关闭。剩下的唯一工作是释放 `ChannelFactory` 通道工厂使用的资源。这一步仅需要调用 `releaseExternalResources()` 方法即可。包括 `NIO Selector` 和线程池在内的所有资源将被自动的关闭和终止。

关闭一个客户端应用是很简单的，但又该如何关闭一个服务端应用呢？你需要释放其绑定的端口并关闭所有接受和打开的连接。为了做到这一点，你需要使用一种数据结构记录所有的活动连接，但这却并不是一件容易的事。幸运的是，这里有一种解决方案，`ChannelGroup`。

`ChannelGroup` 是 Java 集合 API 的一个特有扩展，`ChannelGroup` 内部持有所有打开状态的 `Channel` 通道。如果一个 `Channel` 通道对象被加入到 `ChannelGroup`，如果这个 `Channel` 通道被关闭，`ChannelGroup` 将自动移除这个关闭的 `Channel` 通道对象。此外，你还可以对一个 `ChannelGroup` 对象内部的所有 `Channel` 通道对象执行相同的操作。例如，当你关闭服务端应用时你可以关闭一个 `ChannelGroup` 内部的所有 `Channel` 通道对象。

为了记录所有打开的 `socket`，你需要修改你的 `TimeServerHandler` 实现，将一个打开的 `Channel` 通道加入全局的 `ChannelGroup` 对象，`TimeServer.allChannels`：


Java 代码 

```
275 @Override  
  
276 public void channelOpen(ChannelHandlerContext ctx, ChannelStateEvent e) {  
  
277     TimeServer.allChannels.add(e.getChannel());  
  
278 }
```

代码说明

是的，`ChannelGroup` 是线程安全的。

现在，所有活动的 `Channel` 通道将被自动的维护，关闭一个服务端应用有如关闭一个客户端应用一样简单。

Java 代码 

```
279 package org.jboss.netty.example.time;  
  
280
```

```

281 public class TimeServer {
282
283     static final ChannelGroup allChannels = new DefaultChannelGroup("time-server" );
284
285     public static void main(String[] args) throws Exception {
286         ...
287         ChannelFactory factory = ...;
288         ServerBootstrap bootstrap = ...;
289         ...
290         Channel channel = bootstrap.bind(...);
291         allChannels.add(channel);
292         waitForShutdownCommand();
293         ChannelGroupFuture future = allChannels.close();
294         future.awaitUninterruptibly();
295         factory.releaseExternalResources();
296     }
297 }

```

代码说明

1) `DefaultChannelGroup` 需要一个组名作为其构造器参数。这个组名仅是区分每个 `ChannelGroup` 的一个标示。

2) `ServerBootstrap` 对象的 `bind` 方法返回了一个绑定了本地地址的服务端 `Channel` 通道对象。调用这个 `Channel` 通道的 `close()` 方法将释放这个 `Channel` 通道绑定的本地地址。

- 3) 不管这个 Channel 对象属于服务端，客户端，还是为响应某一个请求创建，任何一种类型的 Channel 对象都会被加入 ChannelGroup。因此，你尽可在关闭服务时关闭所有的 Channel 对象。

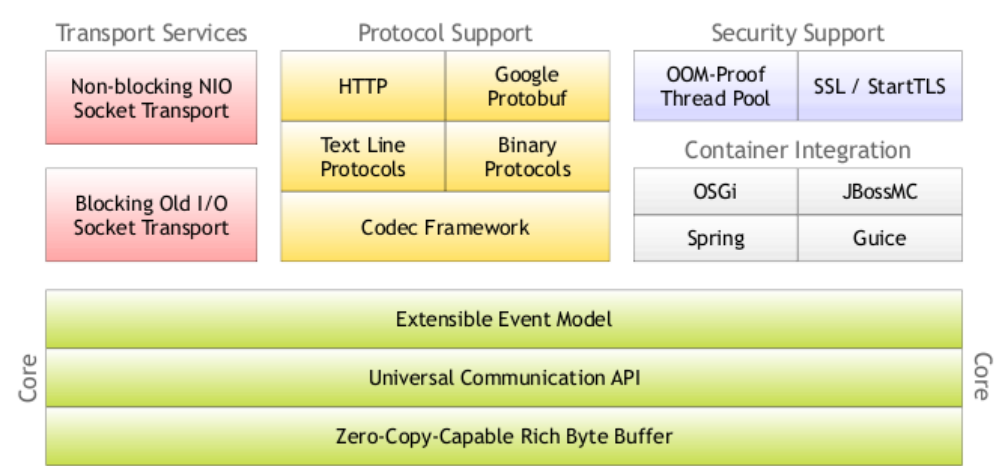
4) waitForShutdownCommand()是一个想象中等待关闭信号的方法。你可以在这里等待某个客户端的关闭信号或者 JVM 的关闭回调命令。

5) 你可以对 ChannelGroup 管理的所有 Channel 对象执行相同的操作。在这个例子里，我们将关闭所有的通道，这意味着绑定在服务端特定地址的 Channel 通道将解除绑定，所有已建立的连接也将异步关闭。为了获得成功关闭所有连接的通知，close()方法将返回一个 ChannelGroupFuture 对象，这是一个类似 ChannelFuture 的对象。

1.10. 总述

在这一章节，我们快速浏览并示范了如何使用 Netty 开发网络应用。下一章节将涉及更多的问题。同时请记住，为了帮助你以及能够让 Netty 基于你的回馈得到持续的改进和提高，[Netty 社区](#) 将永远欢迎你的问题及建议。

第二章. 架构总览



在这个章节，我们将阐述 Netty 提供的核心功能以及在此基础之上如何构建一个完备的网络应用。

2.1. 丰富的缓冲实现

Netty 使用自建的 **buffer API**，而不是使用 NIO 的 **ByteBuffer** 来代表一个连续的字节序列。与 **ByteBuffer** 相比这种方式拥有明显的优势。Netty 使用新的 **buffer 类型 ChannelBuffer**，**ChannelBuffer** 被设计为一个可从底层解决 **ByteBuffer** 问题，并可满足日常网络应用开发需要的缓冲类型。这些很酷的特性包括：

- 如果需要，允许使用自定义的缓冲类型。
- 复合缓冲类型中内置的透明的零拷贝实现。
- 开箱即用的动态缓冲类型，具有像 **StringBuffer** 一样的动态缓冲能力。
- 不再需要调用的 **flip()** 方法。
- 正常情况下具有比 **ByteBuffer** 更快的响应速度。

更多信息请参考：[org.jboss.netty.buffer package description](#)

2.2. 统一的异步 I/O API

传统的 Java I/O API 在应对不同的传输协议时需要使用不同的类型和方法。例如：**java.net.Socket** 和 **java.net.DatagramSocket** 它们并不具有相同的超类型，因此，这就需要使用不同的调用方式执行 **socket** 操作。

这种模式上的不匹配使得在更换一个网络应用的传输协议时变得繁杂和困难。由于（Java I/O API）缺乏协议间的移植性，当你试图在不修改网络传输层的前提下增加多种协议的支持，这时便会产生问题。并且理论上讲，多种应用层协议可运行在多种传输层协议之上例如 **TCP/IP,UDP/IP,SCTP** 和串口通信。

让这种情况变得更糟的是，Java 新的 I/O(NIO)API 与原有的阻塞式的 I/O(OIO)API 并不兼容，NIO.2(AIO)也是如此。由于所有的 API 无论是在其设计上还是性能上的特性都与彼此不同，在进入开发阶段，你常常会被迫的选择一种你需要的 API。

例如，在用户数较小的时候你可能会选择使用传统的 OIO(Old I/O) API，毕竟与 NIO 相比使用 OIO 将更容易一些。然而，当你的业务呈指数增长并且服务器需要同时处理成千上万的客户连接时你便会遇到问题。这种情况下你可能会尝试使用 NIO，但是复杂的 NIO Selector 编程接口又会耗费你大量时间并最终会阻碍你的快速开发。

Netty 有一个叫做 **Channel** 的统一的异步 I/O 编程接口，这个编程接口抽象了所有点对点的通信操作。也就是说，如果你的应用是基于 Netty 的某一种传输实现，那么同样的，你的应用也可以运行在 Netty 的另一种传输实现上。Netty 提供了几种拥有相同编程接口的基本传输实现：

- NIO-based TCP/IP transport (See [org.jboss.netty.channel.socket.nio](#)),
- OIO-based TCP/IP transport (See [org.jboss.netty.channel.socket.oio](#)),
- OIO-based UDP/IP transport, and
- Local transport (See [org.jboss.netty.channel.local](#)).


切换不同的传输实现通常只需对代码进行几行的修改调整，例如选择一个不同的 `ChannelFactory` 实现。

此外，你甚至可以利用新的传输实现没有写入的优势，只需替换一些构造器的调用方法即可，例如串口通信。而且由于核心 `API` 具有高度的可扩展性，你还可以完成自己的传输实现。

2.3. 基于拦截链模式的事件模型

一个定义良好并具有扩展能力的事件模型是事件驱动开发的必要条件。`Netty` 具有定义良好的 I/O 事件模型。由于严格的层次结构区分了不同的事件类型，因此 `Netty` 也允许你在不破坏现有代码的情况下实现自己的事件类型。这是与其他框架相比另一个不同的地方。很多 `NIO` 框架没有或者仅有有限的事件模型概念；在你试图添加一个新的事件类型的时候常常需要修改已有的代码，或者根本就不允许你进行这种扩展。

在一个 `ChannelPipeline` 内部一个 `ChannelEvent` 被一组 `ChannelHandler` 处理。这个管道是[拦截过滤器](#)模式的一种高级形式的实现，因此对于一个事件如何被处理以及管道内部处理器间的交互过程，你都将拥有绝对的控制力。例如，你可以定义一个从 `socket` 读取到数据后的操作：

Java 代码 

```
1 public class MyReadHandler implements SimpleChannelHandler {
2
3     public void messageReceived(ChannelHandlerContext ctx, MessageEvent evt) {
4
5         Object message = evt.getMessage();
6
7         // Do something with the received message.
8
9         ...
10
11        // And forward the event to the next handler.
12
13        ctx.sendUpstream(evt);
14
15    }
16 }
```

同时你也可以定义一种操作响应其他处理器的写操作请求：

Java 代码 

```
11 public class MyWriteHandler implements SimpleChannelHandler {
```



```
12     public void writeRequested(ChannelHandlerContext ctx, MessageEvent evt) {  
13         Object message = evt.getMessage();  
14         // Do something with the message to be written.  
15         ...  
16  
17         // And forward the event to the next handler.  
18         ctx.sendDownstream(evt);  
19     }  
20 }
```

有关事件模型的更多信息，请参考 API 文档 `ChannelEvent` 和 `ChannelPipeline` 部分。

2.4. 适用快速开发的高级组件

上述所提及的核心组件已经足够实现各种类型的网络应用，除此之外，**Netty** 也提供了一系列的高级组件来加速你的开发过程。

2.4.1. Codec 框架

就像“1.8. 使用 POJO 代替 `ChannelBuffer`”一节所展示的那样，从业务逻辑代码中分离协议处理部分总是一个很不错的想法。然而如果一切从零开始便会遭遇到实现上的复杂性。你不得不处理分段的消息。一些协议是多层的（例如构建在其他低层协议之上的协议）。一些协议过于复杂以致难以在一台主机（**single state machine**）上实现。

因此，一个好的网络应用框架应该提供一种可扩展，可重用，可单元测试并且是多层的 **codec** 框架，为用户提供易维护的 **codec** 代码。

Netty 提供了一组构建在其核心模块之上的 **codec** 实现，这些简单的或者高级的 **codec** 实现帮你解决了大部分在你进行协议处理开发过程会遇到的问题，无论这些协议是简单的还是复杂的，二进制的或是简单文本的。

2.4.2. SSL / TLS 支持

不同于传统阻塞式的 I/O 实现，在 NIO 模式下支持 SSL 功能是一个艰难的工作。你不能只是简单的包装一下流数据并进行加密或解密工作，你不得不借助于 `javax.net.ssl.SSLEngine`，`SSLEngine` 是一个有状态的实现，其复杂性不亚于 SSL 自身。你必须管理所有可能的状态，例如密码套件，密钥协商（或重新协商），证书交换以及认证等。此外，与通常期望情况相反的是 `SSLEngine` 甚至不是一个绝对的线程安全实现。

在 Netty 内部，`SslHandler` 封装了所有艰难的细节以及使用 `SSLEngine` 可能带来的陷阱。你所做的仅是配置并将该 `SslHandler` 插入到你的 `ChannelPipeline` 中。同样 Netty 也允许你实现像 [StartTLS](#) 那样所拥有的高级特性，这很容易。

2.4.3. HTTP 实现

HTTP 无疑是互联网上最受欢迎的协议，并且已经有了一些例如 `Servlet` 容器这样的 HTTP 实现。因此，为什么 Netty 还要在其核心模块之上构建一套 HTTP 实现？

与现有的 HTTP 实现相比 Netty 的 HTTP 实现是相当与众不同的。在 HTTP 消息的低层交互过程中你将拥有绝对的控制力。这是因为 Netty 的 HTTP 实现只是一些 HTTP codec 和 HTTP 消息类的简单组合，这里不存在任何限制——例如那种被迫选择的线程模型。你可以随心所欲的编写那种可以完全按照你期望的工作方式工作的客户端或服务端代码。这包括线程模型，连接生命期，快编码，以及所有 HTTP 协议允许你做的，所有的一切，你都将拥有绝对的控制力。

由于这种高度可定制化的特性，你可以开发一个非常高效的 HTTP 服务器，例如：

- 要求持久化链接以及服务器端推送技术的聊天服务（e.g. [Comet](#)）
- 需要保持链接直至整个文件下载完成的媒体流服务（e.g. 2小时长的电影）
- 需要上传大文件并且没有内存压力的文件服务（e.g. 上传1GB 文件的请求）
- 支持大规模 mash-up 应用以及数以万计连接的第三方 web services 异步处理平台

2.4.4. Google Protocol Buffer 整合

[Google Protocol Buffers](#) 是快速实现一个高效的二进制协议的理想方案。通过使用 `ProtobufEncoder` 和 `ProtobufDecoder`，你可以把 Google Protocol Buffers 编译器 (protoc)生成的消息类放入到 Netty 的 codec 实现中。请参考“[LocalTime](#)”实例，这个例子也同时显示出开发一个由[简单协议定义](#) 的客户及服务端是多么的容易。

2.5. 总述

在这一章节，我们从功能特性的角度回顾了 Netty 的整体架构。Netty 有一个简单却不失强大的架构。这个架构由三部分组成——缓冲（buffer），通道（channel），事件模型（event model）——所有的高级特性都构建在这三个核心组件之上。一旦你理解了它们之间的工作原理，你便不难理解在本章简要提及的更多高级特性。

你可能对 **Netty** 的整体架构以及每一部分的工作原理仍旧存有疑问。如果是这样,最好的方式是[告诉我们](#) 应该如何改进这份指南。