Netty 3.2 用户手册

快速有效的网络应用开发

3.2.4.Final 译者：张立明 Larry Zhang 二次整理：Princy

前言

1. 问题提出

当前我们使用通用的应用或库来相互通信。比如，我们常常使用 HTTP 客户端库来从 WEB 服务器上获取信息，并通过 Web Service 来调用一个远程过程。

然而，一个通用的协议或者它的实现，有时候并不能很好的扩展。这一点类似于我们不适用通用的 HTTP 服务器来交换大的文件、电子邮件和诸如财务信息和多人游戏数据等近乎于实时的数据。这些东西需要根据其特定用途而进行高度优化的协议实现。例如，你可能需要一个专门针对基于 AJAX 的聊天应用、针对多媒体流、或者大的文件传输进行了优化的 HTTP 服务器。你甚至可能想设计并实现一个完全按照你的需求而定义的全新的协议。

另一种情况也是难以避免的。那就是，为了和一个既有的旧系统进行交互，你必须处理旧系统上使用的协议。这时，在不牺牲稳定性和性能的前提下，你能够在多长时间内实现那个协议就非常重要。

2. 解决方案

*Netty 项目*是一个提供异步的、事件驱动的网络应用框，是一套有助于快速开发出高性能、高扩展性的、高可维护性的协议的服务器或客户端的开发工具。

换而言之，Netty 是一个基于 NIO 的 C/S 框架。这套框架可以快速、简单地开发出网络协议的客户端和服务器端应用。它可以大大简化、流程化 TCP 和 UDP 套接字的服务器开发过程。

“快速和简单”并不意味着开发出的应用会遇到可维护性、性能等问题。Netty 是建立在从许多网络协议（如 FTP、SMTP、HTTP 和各种二进制和文本协议等）中借鉴的经验基础上精心设计出的。这使得 Netty 在开发的简单化、性能、稳定性、灵活性等方面都同时达到了设计目标。

一些用户可能已经发现了其他的一些网络应用框架。这些框架也宣称具有相同的优势。这时你可能会问：Neey 有什么不同？答案是“道不同”。Netty 设计的原则是：给你提供从 API 到实现以最舒适的体验。这一点是看不到摸不着的。但你在阅读这个文档、以及应用 Netty 过程中，你会体验到我们的这个设计原则使得一切变得轻松容易。

Chapter 1. 开始

这一章围绕着 Neey 的核心构成讲述，并提供了简单的例子以便快速上手。读到本章末尾，你将可以写一个基于 Netty 的客户端和服务器。

如果你喜欢自顶向下的学习方式，你应该从 Chapter 2, *架构概览*开始，然后再回到这里。

1.1. 开始之前

运行本章中的例子最低的要求只有两个：最新版本的 Netty 和 JDK1.5 或更高版本。最新的 Netty 可以在此下载。要下载到正确的 JDK 版本，请参考你选择的 JDK 提供商的网站。

在读的过程中，你会对本章中涉及的类有更多的疑问。当你想了解更多的时候，请参考 API 文档。所有的类名，都非常方便地连接到了在线的 API 页面上。此外，记得联系我们 Netty 社区并告诉我们是否有些信息不正确，语法或者拼写等错误，或者你有一个提高这个文档的好办法。

1.2. 编写一个 Discard 服务

这个世界上最简单的协议不是"Hello，World！"，而是 DISCARD。这个协议丢弃所有的收到的数据，不给任何回应。

为了实现这个 DISCARD 协议，你唯一要做的事情就是忽略所有收到的数据。我们直接从处理器（handler）的实现开始。这个处理器处理 Netty 生成的 I/O 事件。

```
packageorg.jboss.netty.example.discard;


public class DiscardServerHandler extends SimpleChannelHandler {(1)


    @Override

public void messageReceived(ChannelHandlerContextctx, MessageEvent e) {(2)

    }


    @Override

public void exceptionCaught(ChannelHandlerContextctx, ExceptionEvent e) {(3)

e.getCause().printStackTrace();


Channelch = e.getChannel();

ch.close();

    }
```

```
}
```

1. DiscardServerHandler 继承 SimpleChannelHandler，ChannelHandler 的接口实现类。

   SimpleChannelHandler 提供各种事件的处理方法，你可以重载它们。到目前为止，继承 SimpleChannelHandler，而不是你自己去实现一个 handler 接口，是足够的。

2. 我们在这里重载了 messageReceived 事件处理方法。这个方法调用时提供了 MessageEvent，它包含着刚刚从客户端收到的新数据。在这个例子中，我们通过什么也不做，来忽略收到的数据，从而实现 DISCARD 协议。

3. 当一个异常因为 I/O 错误由 Netty 抛出，或者由在处理事件过程中，handler 的实现抛出了异常，exceptionCaught 方法会被调用，并提供了 ExceptionEvent。尽管在特定情况下，实现这个方法时，你需要对异常有不同的处理，但通常情况下，被捕获的异常应该被记录，并且对应的 channel 应该被关闭。例如，你可能想在关闭链接之前发送一个错误代码的回应信息。

到此为止，我们已经实现了 DISCARD 服务的一半。接下来需要写 main 方法来运行这个配备了 DiscardServerHandler 的服务。

```
packageorg.jboss.netty.example.discard;


importjava.net.InetSocketAddress;

importjava.util.concurrent.Executors;


public class DiscardServer {


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

ChannelFactory factory =

newNioServerSocketChannelFactory(1)(

Executors.newCachedThreadPool(),

Executors.newCachedThreadPool());
```

```
ServerBootstrap bootstrap = new ServerBootstrap(factory);


bootstrap.setPipelineFactory(new ChannelPipelineFactory() {

publicChannelPipelinegetPipeline() {

returnChannels.pipeline(new DiscardServerHandler());

            }

        });


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

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


bootstrap.bind(new InetSocketAddress(8080));

    }

}
```

1. ChannelFactory 是创建并管理 Channel 和它们相关资源的工厂。它处理所有的 I/O 请求，执行 I/O 来生成 ChannelEvent。Netty 提供多种 ChannelFactory 实现。我们现在正在实现一个服务器端的例子，因此我们使用 NioServerSocketChannelFactory。另一个需要知道的是，它并不是自行创建 I/O 线程。它试图从你在构造方法中指定的线程池中获得线程。对于线程是如何在你的应用运行环境中去管理的，它给了更多的控制，比如一个具有安全管理机制的应用服务器。

2. ServerBootstrap 是一个建立服务器的工具类。你当然可以直接使用 Channel 来构建一个服务器，但你要清楚这将是一个繁琐的过程，而其实你根本没必要这么做。

3. 这里，我们配置了 ChannelPipelineFactory。当一个新的连接接入到服务器，一个新的 ChannelPipeline 将由指定的 ChannelPipelineFactory 来创建。这个新的 Pipeline 包含着 DiscardServerHandler。随着这个应用逐步完善，最终实际你就是添加更多的 handler 到 Pipeline，并抽象出这个匿名类成为一个顶级类。

4. 你还可以设置针对 Channel 实现的特定参数。我们正在编写的是 TCP/IP 服务，所以我们可以设置套接字的选项参数，如 tcpNoDelay 和 keepAlive。请注意到这个"child."前缀出现在所有参数前，它

意味着这个选项参数应用于接入的 Channel，而不是 ServerSocketChannel 的参数。你可以按下面做法来为 ServerSocketChannel 设定参数。

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

5. 快要可以运行了。接下来要做的是绑定端口并启动服务。这里我们绑定所有本机网卡的端口 8080。你可以用不同的绑定地址来多次调用 bind 方法。

哈哈！我们在 Netty 上构建了第一个服务器应用。

1.3. 详解 Received Data

刚才我们已经写了我们第一个服务器。现在需要的是测试一下它的运行情况。最简单的测试方法，莫过于使用 telnet 命令了。例如，你可以在命令行上输入**"telnet localhost 8080"**，然后随便输入些什么。

问题是，我们能说这个服务器运行正常吗？很难说的，因为它是一个"丢弃"服务，根本没有任何回应。为了证实它却是运转正常，我们改一下这个服务，让它打印出收到的数据。

我们已经知道了，当收到了数据时，会生成 MessageEvent，还会调用 messageReceived 处理器方法。我们可以在 DiscardServerHandler 的 messageReceived 中加入代码：

```
@Override
public void messageReceived(ChannelHandlerContextctx, MessageEvent e) {
ChannelBuffer(1) buf = (ChannelBuffer) e.getMessage();
while(buf.readable()) {
System.out.println((char) buf.readByte());
System.out.flush();
    }
}
```

1. 在套接字中传递的消息永远都是 ChannelBuffer。ChannelBuffer 是一个核心的数据结构，它存储着 Netty 中的字节序列。它很像 NIO 中的 ByteBuffer，不过更加简单和灵活了。例如，Netty 允许你

构建一个由多个 ChannelBuffer 复合而成的 ChannelBuffer，以减少不必要的内存复制。

尽管它在很多方面都和 NIO 的 ByteBuffer 相像，仍然建议参考一下 API 手册。学习如何正确使用 ChannelBuffer 是使用轻松驾驭 Netty 的重要一步。

如果你再次运行 **telnet** 命令，你会看到服务器打印出它收到的内容。

"丢弃"服务器的全部源代码位于 org.jboss.netty.example.discard 包。

1.4. 编写一个 Echo 服务

截至目前，我们已经实现了数据的获取，但没有回应。实际上，一个服务器应该对请求给予回应的。我们看一下如何通过实现 ECHO 协议给客户端一个回应，把收到的数据送回去。

这和我们在上一节中实现的"丢弃"服务之间唯一不同的是它把收到的数据又发送给回客户端，而不是在服务器端打印出来。为此，修改一下 messageReceived 方法就可以了：

```
@Override
public void messageReceived(ChannelHandlerContextctx, MessageEvent e) {
Channel ch = e.getChannel();
ch.write(e.getMessage());
}
```

1 ChannelEvent 对象有一个相应的 Channel 的引用。这里返回的 Channel 代表收到消息事件 MessageEvent 的那个连接。我们可以得到这个 Channel 调用它的 write 方法来写数据给对等的远端。

如果你再次运行 **telnet**，你会看到服务器把你发给它的都送了回来。

Echo 服务器的全部代码位于 org.jboss.netty.example.echo 包。

1.5. 编写一个 Time 服务

这小节我们要实现的协议是 TIME。和上个例子不同，它发送一个包含 32 位证书的消息，发送完毕后不需要收到任何回应就关闭连接。在这个例子中，你将学习如何构建并发送一个消息，然后关闭连接。

连接建立后，收到的任何数据都被忽略不计，而仅仅是发送一个消息。因此，我们这次不能使用 messageReceived 方法，而是需要重载 channelConnected 方法。代码如下：

```
packageorg.jboss.netty.example.time;


public class TimeServerHandler extends SimpleChannelHandler {


    @Override

public void channelConnected(ChannelHandlerContextctx, ChannelStateEvent e) {(1)

Channelch = e.getChannel();


ChannelBuffer time = ChannelBuffers.buffer(4);(2)

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


ChannelFuture f = ch.write(time);(3)


f.addListener(new ChannelFutureListener() {(4)

public void operationComplete(ChannelFuture future) {

Channelch = future.getChannel();

ch.close();

        }

    });

}


    @Override

public void exceptionCaught(ChannelHandlerContextctx, ExceptionEvent e) {
```

```
    e.getCause().printStackTrace();

    e.getChannel().close();

        }

    }
```

1. 我们前面解释过，channelConnected 方法在连接建立的时候被调用。我们在这里发送 32 位整数来表示当前时间（单位：秒）。

2. 为了发送新消息，需要分配一个用来包含消息的缓存。我们需要发送的是 32 位整数，所以我们需要一个 4 字节容量的 ChannelBuffer。这个 ChannelBuffers 工具类用来分配新的缓存。除了这个 buffer 方法，ChannelBuffers 提供了很多和 ChannelBuffer 相关的有用的方法，请参考 API 手册。

   此外，静态导入 ChannelBuffers 是一种好的做法：

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

...

ChannelBufferdynamicBuf = dynamicBuffer(256);

ChannelBufferordinaryBuf = buffer(1024);
```

3. 一般而言，我们编写结构化的消息。

   但等一下，flip 呢？我们过去在 NIO 中发送一个消息之前，不是调用 ByteBuffer.flip()吗？因为 ChannelBuffer 有两个指针，所以没有这个方法。一个指针用于读操作，一个用于写操作。这个写操作的索引在你向 ChannelBuffer 中写入内容时增加，同时读操作的索引不改变。这个读写的索引分别表示消息开始和结束的位置。

   相反，不调用 flip 方法的话，NIO 缓存不提供一个清晰的方式来搞清一个消息内容的起始位置。如果你忘记调用 flip 的话，你会遇到麻烦：错误的数据发出，或者什么也不发出。因为不同的操作类型有不同的指针，所以这种情况对 Netty 而言是不会出现的。你会发现你在这个不需要 flip 的环境中，非常的适应、非常舒服。

   另一个需要明确的是这个 write 方法返回一个代表尚未发生的后续 I/O 操作的 ChannelFuture。这意味着，因为 Netty 中的所有操作都是异步的，调用的任何操作都可能尚未真的执行。比如下面的代

码甚至在一个消息尚未发出前关闭连接：

```
Channelch = ...;

ch.write(message);

ch.close();
```

因此，你需要在 write 方法返回的 ChannelFuture 提醒你写操作完成之后，调用 close 方法。请注意，close 方法同样也可能不是立即关闭，它也返回 ChannelFuture。

4. 那么当 write 调用完成的时候，我们怎么获得提醒？简单的给返回的 ChannelFuture 增加一个 ChannelFutureListener 即可。这里我们构建一个新的匿名的 ChannelFutureListener 以实现在调用完成后关闭 Channel。

另外一种做法，你可以使用一个预定义好的 Listener 来简化代码：

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

1.6. 编写 Time 客户端

和 DISCARD、ECHO 服务不同，因为人类不能转译 32 位整数为日历时间，所以我们需要一个 TIME 协议的客户端。本节中我们讨论如何确保服务器正确工作，并学习如何用 Netty 写一个客户端。

在 Netty 中，服务器端和客户端最大的、唯一的不同是需要不同的 Bootstrap 和 ChannelFactory。请看一下下面代码：

```
packageorg.jboss.netty.example.time;

importjava.net.InetSocketAddress;

importjava.util.concurrent.Executors;


public class TimeClient {
```

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

        String host = args[0];

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



ChannelFactory factory =

newNioClientSocketChannelFactory(1)(

Executors.newCachedThreadPool(),

Executors.newCachedThreadPool());



ClientBootstrap bootstrap = new ClientBootstrap(2)(factory);



bootstrap.setPipelineFactory(new ChannelPipelineFactory() {

publicChannelPipelinegetPipeline() {

returnChannels.pipeline(new TimeClientHandler());

        }

    });



bootstrap.setOption("tcpNoDelay"(3), true);

bootstrap.setOption("keepAlive", true);



bootstrap.connect(4)(new InetSocketAddress(host, port));

    }

}
```

1. 创建客户端的 Channel 使用的是 NioClientSocketChannelFactory，而不是 NioServerSocketChannelFactory。
2. ClientBootstrap 在客户端，对应服务器端的 ServerBootstrap。
3. 请注意到，这次没有"child."前缀。客户端的 SocketChannel 没有上一级根。

4 应该调用 connect 方法而不是 bind 方法。

我们能看出，这跟服务器端的启动过程没有特别的区别。那么，ChannelHandler 实现呢？它应该接收 32 位整数，并将其解释为人可读的格式，打印解释出来的时间，然后关闭连接：

```
packageorg.jboss.netty.example.time;

importjava.util.Date;

public class TimeClientHandler extends SimpleChannelHandler {

    @Override
public void messageReceived(ChannelHandlerContextctx, MessageEvent e) {
ChannelBufferbuf = (ChannelBuffer) e.getMessage();
longcurrentTimeMillis = buf.readInt() * 1000L;
System.out.println(new Date(currentTimeMillis));
e.getChannel().close();
    }

    @Override
public void exceptionCaught(ChannelHandlerContextctx, ExceptionEvent e) {
e.getCause().printStackTrace();
e.getChannel().close();
    }
}
```

看上去真的很简单，并且和服务器端代码没什么区别。不过，这个 handler 时常会不工作，而是抛出 IndexOutOfBoundsException。我们在下一节讨论为什么会是这样。

1.7. 处理基于流的传输

1.7.1. 套接字缓存的一个警示

在诸如 TCP/IP 这样的基于流的传输机制下，收到的数据存储在套接字的接收缓存中。不幸的是，流传输的缓存并不是"包"的队列，而是"字节"的队列。这意味着，就算你以两个独立的包的形式发送两条消息，操作系统不会按两条消息来处理他们，而是一系列的字节。因此，没有任何机制能保证你读到的就是对端写入的。比如，我们假定操作系统的 TCP/IP 栈收到了下面三个包：

```
+-----+-----+-----+

| ABC | DEF | GHI |

+-----+-----+-----+
```

基于流的协议的通用特性导致你的应用程序按下面的片段来读到数据的几率非常之高：

```
+----+-------+---+---+

| AB | CDEFG | H | I |

+----+-------+---+---+
```

因此，一个接收方（无论是服务器端还是客户端）应该能够将收到的数据构造成一个或多个对你的应用程序逻辑而言有意义的、易于理解的"帧"。就上面的例子而言，接收到的数据应该被重组成为下面的情况：

```
+-----+-----+-----+

| ABC | DEF | GHI |

+-----+-----+-----+
```

1.7.2. 第一个解决办法

好，我们现在回到 TIME 例子。我们这里也有同样的问题。一个 32 位整数是一个很少量的数据，不常被拆分。但问题是，它是可以被拆分的，而且随着流量的增加，这种拆分的可能性会加大。

最简单的解决办法应该是建立一个内部的汇聚缓存，并且等待直至 4 个字节都收到进入了内部缓存中。下面是修改后的 TimeClientHandler 实现，它解决了这个问题：

```
packageorg.jboss.netty.example.time;


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

importjava.util.Date;

public class TimeClientHandler extends SimpleChannelHandler {

private final ChannelBufferbuf = dynamicBuffer();(1)

    @Override
public void messageReceived(ChannelHandlerContextctx, MessageEvent e) {

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

buf.writeBytes(m);(2)


if (buf.readableBytes() >= 4) {(3)

longcurrentTimeMillis = buf.readInt() * 1000L;

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

e.getChannel().close();

        }

    }


    @Override

public void exceptionCaught(ChannelHandlerContextctx, ExceptionEvent e) {

e.getCause().printStackTrace();
```

```
e.getChannel().close();

    }

}
```

1. 这个动态的缓存是能够根据需要增加容量的 ChannelBuffer。这在我们不知道消息的大小的时候，非常有用。
2. 首先，所有收到的数据都必须汇聚到 buf 中。
3. 然后，这个 handler 必须检查 buf 中是否有足够的数据--在这里是 4 个字节。接下来按业务逻辑处理。否则，Netty 继续在后续数据到达时调用 messageReceived 方法，直至最终收集到 4 个字节。

1.7.3. 第二个解决办法

尽管第一个解决办法确实解决了 TIME 客户端的问题，修改后的 handler 看上去已经不是那么简洁了。想象一下更加复杂的协议，协议涉及多个字段、可变长的字段。你的 ChannelHandler 实现会很快变得很难维护。

你可能也注意到，你可以增加多个 ChannelHandler 给 ChannelPipeline，由此，你可以拆分单一的 ChannelHandler 为多个模块化的 handler 来减少你的应用程序的复杂度。例如，你可以拆分 TimeClientHandler 为两个 handler：

- TimeDecoder 处理字节重组的问题，和
- 最初那个简单的 TimeClientHandler.

幸运的是，Netty 提供了可扩展的类来帮助你写出第一个类：

```
packageorg.jboss.netty.example.time;


public class TimeDecoder extends FrameDecoder 1 {


    @Override

protected Object decode(

ChannelHandlerContextctx, Channelchannel, ChannelBuffer buffer) 2 {
```

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

return null; 3

        }


returnbuffer.readBytes(4); 4

    }

}
```

1. FrameDecoder 是 ChannelHandler 的一个实现，使得处理重组问题简单。
2. FrameDecoder 调用 decode 方法，收到新的数据时，内部有一个汇聚的缓存。
3. 如果返回的是 null 以为这收到的数据还不够。FrameDecoder 会在数据量足够的时候再次调用。
4. 如果返回的不是 null 意味着 decode 方法已经成功解码了一个消息。FrameDecoder 会丢弃位于汇聚缓存的已经读走的数据。记住，你不需要解码多个消息，因为 FrameDecoder 会一直调用 decoder 直至返回 null。

现在我们已经有了另一个 handler 可以插入到 ChannelPipeline，我们应该修改 TimeClient 的 ChannelPipelineFactory 实现：

```
bootstrap.setPipelineFactory(new ChannelPipelineFactory() {

publicChannelPipelinegetPipeline() {

returnChannels.pipeline(

newTimeDecoder(),

newTimeClientHandler());

        }

    });
```

如果你喜欢尝试新的东西，你应该乐意试一下 ReplayingDecoder。它更大程度的简化了解码器。请参看 API 手册以获得更多信息。

```
packageorg.jboss.netty.example.time;


public class TimeDecoder extends ReplayingDecoder<VoidEnum> {


    @Override
protected Object decode(

ChannelHandlerContextctx, Channelchannel,

ChannelBuffer buffer, VoidEnum state) {



returnbuffer.readBytes(4);

    }

}
```

此外，Netty 提供了一些"开箱即用"的解码器，它们可以使你很容易地实现很多协议，帮助你避免做出一个难以维护的、单一的 handler 实现。请参看下面的包，以获得更多信息：

- org.jboss.netty.example.factorial 针对的是二进制协议
- org.jboss.netty.example.telnet 针对的是基于行的文本协议

1.8. 用 POJO 替代 ChannelBuffer

到此为止，我们所有的例子都在使用 ChannelBuffer 作为基础的协议消息数据结构。本节中，我们使用 POJO 替代 ChannelBuffer 来改进 TIME 协议的客户端和服务器端的例子。

ChannelHandler 中使用 POJO 的优点是很明显的。从 handler 中分离出那些从 ChannelBuffer 提取信息的代码，使得 Handler 变得更加可维护、可复用。在这个 ITME 客户端、服务器端的例子中，我们只读取 32 位整数，所以直接使用 ChannelBuffer 没什么大问题，但在实际开发协议中，你会发现这种分离是十分必要的。

首先，我们定义一个新的类型 UnixTime。

```
packageorg.jboss.netty.example.time;


importjava.util.Date;


public class UnixTime {
private final int value;


publicUnixTime(int value) {

this.value = value;

    }


publicintgetValue() {

return value;

    }


    @Override

public String toString() {

return new Date(value * 1000L).toString();

    }

}
```

现在修改一下 TimeDecoder，不再返回 ChannelBuffer，而是返回 UnixTime。

```
@Override

protected Object decode(

ChannelHandlerContextctx, Channelchannel, ChannelBuffer buffer) {

if (buffer.readableBytes() < 4) {

return null;
```

```
    }


return new UnixTime(buffer.readInt());(1)

}
```

(1) FrameDecoder 和 ReplayingDecoder 允许你返回任何类型的对象。如果你只能返回 ChannelBuffer 的话，那么我们必须添加另外一个 ChannelHandler 来把 ChannelBuffer 转换为 UnixTime。

修改了解码的方法后，TimeClientHandler 不再使用 ChannelBuffer 了：

```
@Override

public void messageReceived(ChannelHandlerContextctx, MessageEvent e) {

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

System.out.println(m);

e.getChannel().close();

}
```

是不是看上去更简单、更优雅了？同样的技术可以用于服务器端。这次我们先更新一下 TimeServerHandler：

```
@Override

public void channelConnected(ChannelHandlerContextctx, ChannelStateEvent e) {

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

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

f.addListener(ChannelFutureListener.CLOSE);

}
```

现在唯一缺少的就是编码器了。这个编码器应该是一个把 ChannelBuffer 转换为 UnixTime 的 ChannelHandler 实现。编码一条消息时，因为不需要处理数据包的拆解和重组，所以这比写一个解码器要简单得多。

```
packageorg.jboss.netty.example.time;


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


public class TimeEncoder extends SimpleChannelHandler {


public void writeRequested(ChannelHandlerContextctx, MessageEvent (1) e) {

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


ChannelBufferbuf = buffer(4);

buf.writeInt(time.getValue());


Channels.write(ctx, e.getFuture(), buf); (2)

    }

}
```

(1) 编码器重载了 writeRequested 方法来拦截一个写入的请求。请注意 MessageEvent 参数和在 messageReceived 的是同一类型，但是却有不同的解读。一个 ChannelEvent 可以是上行或下行事件，这取决于事件的传递方向。比如，MessageEvent 在 messageReceived 中是上行事件，但是在 writeRequested 中却是下行事件。请参考 API 文档以获得关于上行事件和下行事件之间的更多区别。

(2) 一旦完成从 POJO 到 ChannelBuffer 的转换，你应该将这个新的缓存提交给 ChannelPipeline 中的前一个 ChannelDownstreamHandler。Channels 提供了各种工具性的方法来生成和发送一个 ChannelEvent。本例中，Channels.write(...)方法创建了一个新的 MessageEvent 并发送给 ChannelPipeline 中的前一个 ChannelDownstreamHandler。

此外，建议使用静态导入的方式使用 Channels：

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

...
```

```
ChannelPipeline pipeline = pipeline();

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

fireChannelDisconnected(ctx);
```

最后的工作就是给服务器端的 ChannelPipeline 添加 TimeEncoder 方法。我们把这个留作练习吧。

1.9. 关闭应用程序

如果运行 TimeClient，需要注意到，这个应用程序会什么也不做一直运行下去而不退出。从完整的栈跟踪信息看，可以看到一些 I/O 线程正在运行。为了关闭这些 I/O 线程，使得应用程序优雅地关闭退出，我们需要释放由 ChannelFactory 分配的资源。

关闭一个典型的网络应用的过程由下面三步组成：

1. 关闭所有的服务器套接字。
2. 关闭所有的非服务器套接字（如，客户端套接字和受理的套接字）。
3. 释放所有 ChannelFactory 的资源。

如果在 TimeClient 应用这三步，关闭唯一的客户端连接，释放所有 ChannelFactory 持有的资源，TimeClient.main()就可以很完整地关闭了：

```
packageorg.jboss.netty.example.time;



public class TimeClient {

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

        ...

ChannelFactory factory = ...;

ClientBootstrap bootstrap = ...;

        ...

ChannelFuture future(1) = bootstrap.connect(...);

future.awaitUninterruptibly();(2)
```

```
if (!future.isSuccess()) {

future.getCause().printStackTrace();(3)

        }

future.getChannel().getCloseFuture().awaitUninterruptibly();(4)

factory.releaseExternalResources();(5)

    }

}
```

(1) ClientBootstrap 的 connect 方法返回了 ChannelFuture，后者在连接成功或失败的时候能给出提醒。此外，它还有一个对试图建立的连接相关的 Channel 的引用。

(2) 等待返回的 ChannelFuture 以确定建立连接的尝试是否成功。

(3) 如果连接建立失败，打印出失败的原因。如果建立连接的尝试既没有成功，也不是取消了，那么这个 ChannelFuture 的 getCause()方法会返回具体的失败原因。

(4) 现在已经完成了建立连接的尝试。我们现在需要等待，等待 Channel 的 closeFuture 方法返回，直到这个连接被关闭。每个 Channel 都有它自己的 closeFuture，你可以在连接关闭的时候得到提醒，并执行一些特定的动作。

当连接的尝试失败时，Channel 会被自动关闭，因此即便这个连接尝试是失败 closeFuture 仍然会被调用。

(5) 到这里，所有的连接都已经关闭了。遗留的唯一任务是释放 ChannelFactory 占用的资源。调用一下它的 releaseExternalResources()就可以了。所有的资源，包括 NIO 的 Selector 和线程池将被自动关闭终止。

关闭客户端是非常容易的事儿，但关闭服务器端呢？这需要关闭所有建立的连接、释放端口占用。为此，需要一个记录所有活动连接的数据结构，这可不是琐碎的小事儿一桩。还好，有一个办法，那就是 ChannelGroup。

ChannelGroup 代表打开的 Channel 的集合，是 Java collection API 的特殊扩展。如果一个 Channel 加入了 ChannelGroup，然后这个 Channel 被关闭了，那么这个关闭的 Channel 会自动从它的 ChannelGroup

中移除。你可以对一个组执行一个操作，却作用于所有属于同一个组的 Channel。比如，你可以在停止服务器的时候，关闭一个 ChannelGroup 内所有的 Channel。

为了记录跟踪到所有打开的套接字，你需要修改 TimeServerHandler 来给 ChannelGroup(TimeServer.allChannels)添加一个新的 Channel:

```
@Override

public void channelOpen(ChannelHandlerContextctx, ChannelStateEvent e) {

TimeServer.allChannels.add(e.getChannel());(1)

}
```

(1) 没错，ChannelGroup 是线程安全的。

这下所有的启用 Channel 都自动的维护起来了。关闭服务器变得和关闭客户端一样简单了:

```
packageorg.jboss.netty.example.time;


public class TimeServer {


static final ChannelGroupallChannels = new DefaultChannelGroup("time-server"(1));


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

        ...

ChannelFactory factory = ...;

ServerBootstrap bootstrap = ...;

        ...

Channelchannel(2) = bootstrap.bind(...);

allChannels.add(channel);(3)

waitForShutdownCommand();(4)

ChannelGroupFuture future = allChannels.close();(5)
```

```
future.awaitUninterruptibly();

factory.releaseExternalResources();

    }

}
```

1. DefaultChannelGroup 需要一个名字作为构造方法的参数。这个名字仅仅是用来区别于其他组。

2. 这个 ServerBootstrap 的 bind 方法返回了一个服务器端的、绑定了指定本地地址的 Channel。调用返回的 Channel 的 close()方法可以使得它和绑定的本地地址解除绑定。

3. 无论是服务器的、客户端的还是接受的 Channel，都可以添加到 ChannelGroup。因此你可以在关闭服务器的时候，一下子就关闭关闭绑定的 Channel 和接受的 Channel。

4. waitForShutdownCommand()方法等待关闭的指令，它是一个假定会发生的方法，等待从授权的客户端或 JVM 关闭钩子发过来的消息。

5. 可以对位于同一个 ChannelGroup 的 channel 执行相同的操作。本例中，我们关闭所有的 channel，包括绑定的服务器 Channel 会解除绑定，所有的受理的连接会被异步地关闭。为了得到所有连接关闭的提醒，它返回了 ChannelGroupFuture，这和 ChannelFuture 的角色几乎相近。

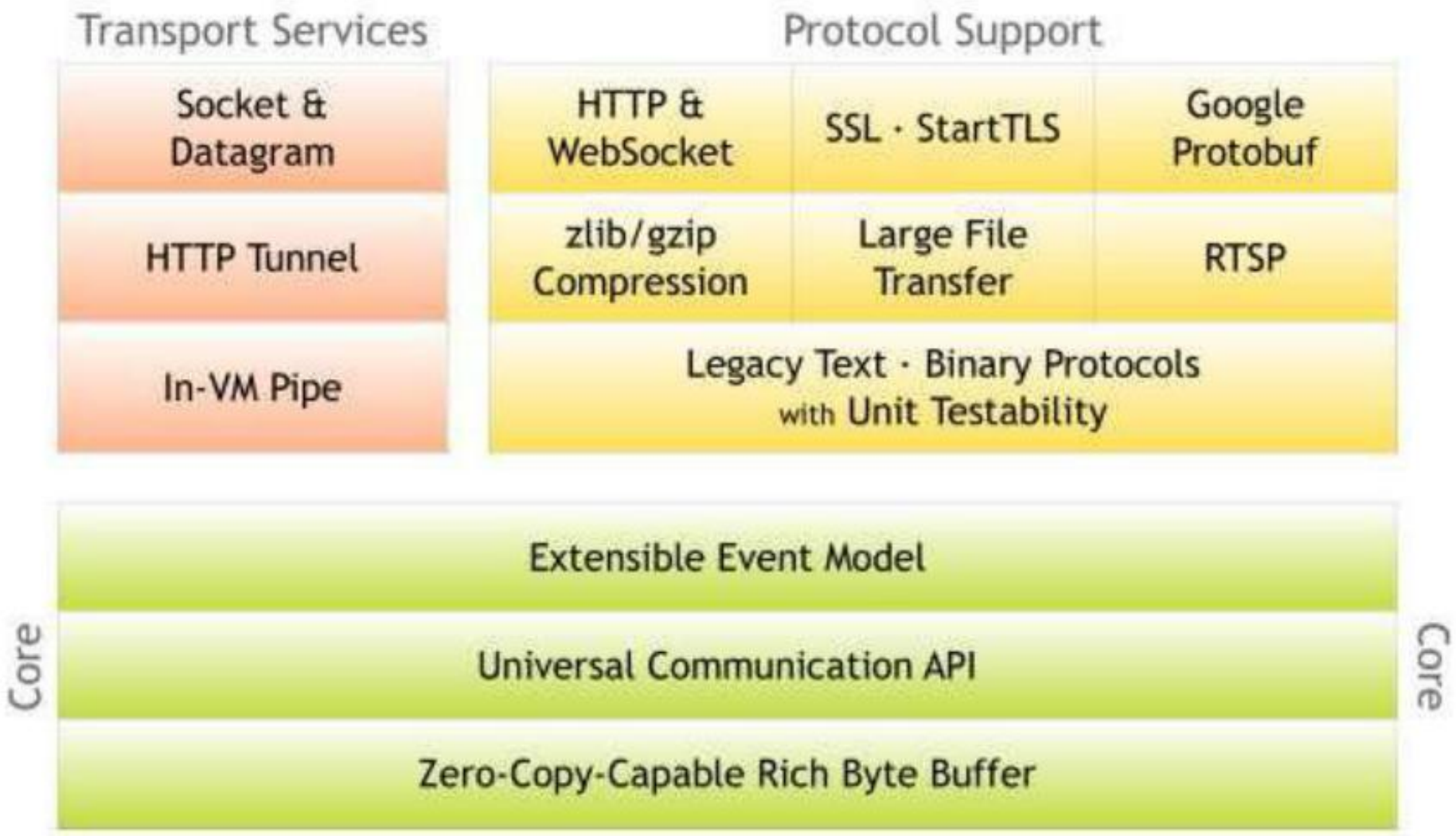
1.10. 总结

本章中，我们通过一个关于如何基于 Netty 写一个网络应用的演示例子，快速浏览了 Netty。后续的章节和修订后的本章会提及你可能有疑问的一些内容。要知道讨论区一直都在期待你的问题、建议来帮助我们根据你的反馈而持续改进 Netty。

Chapter 2. 架构概览

2.4.4. Google Protocol Buffer 集成

2.5. 总结



本章中，我们会看一下 Netty 中提供了哪些核心功能，它们是如何组成了一个完整的网络应用开发框架的。

在阅读本章的过程中，请始终记得这张图。

2.1. 丰富的 Buffer 数据结构

Netty 用它自己的独有的 buffer API，而不是 NIO 的 ByteBuffer 来表示一系列的字节。这种做法比使用 ByteBuffer 具有很明显的优势。Netty 的新 buffer 类型，ChannelBuffer 专门设计用来解决 ByteBuffer 的缺陷，并满足网络应用开发人员的日常开发需要。下面列出了一些酷的特性：

- 如果必要，你可以定义你的 buffer 类型。
- 通过内建的复合 buffer 类型，实现了透明的零拷贝。
- 提供了一个和 StringBuffer 类似的、容量可以根据需要扩展的、动态的 buffer 类型。
- 再也不需要调用 flip()了。
- 比 ByteBuffer 速度快。

更多信息，请参考 org.jboss.netty.buffer 包描述。

2.2. 统一的异步 I/O API

Java 中传统的 I/O API 为不同的传输类型提供了不同的类型和方法。例如，java.net.Socket 和 java.net.DatagramSocket 并不具有任何共同的父类型，因此它们在执行套接字 I/O 的方式完全不同。

这使得一个网络应用如果要从一种传输方式转换为另一种方式变得非常困难和琐碎。这种在传输上不能转换的特性，在你需要支持各种传输方式，却不能重写整个网络层的时候，就是一个问题了。逻辑上，很多协议是可以运行于多种传输方式的，如 TCP/IP、UDP/IP、SCTP、和串口通信。

更加糟糕的是，Java New I/O (NIO) API 引入了和原有的阻塞 I/O(OIO) API 之间的不兼容。并且这种不兼容还会出现在 NIO.2 (AIO)。因为这些 API 之间在设计和性能特性上都不同，你必须在开始实现的时候就确定你的而应用程序基于哪套 API。

例如，因为一些客户端程序是非常小的，并且用 OIO 写一个服务器会比用 NIO 简单很多，所以你可能想从 OIO 开始。然而，当你的业务按指数级成长，你的服务器开始同时为几万个客户端服务的时候，你的麻烦来了。你也可能想从 NIO 开始，但考虑到 NIO Slector API 的复杂度对于快速开发的影响，整个开发周期会长很多。

Netty 有一个统一的异步 I/O 接口，名为 Channel，它抽象出了点对点通信需要的所有操作。这意味着，一旦你基于一种 Netty 传输开发了应用，你的应用还可以运行于其他 Netty 传输方式上。Netty 通过一致的 API 提供了很多重要的传输方式：

- 基于 NIO 的 TCP/IP 传输（参看 org.jboss.netty.channel.socket.nio),
- 基于 OIO 的 TCP/IP 传输（参看 org.jboss.netty.channel.socket.oio),
- 基于 OIO 的 UDP/IP 传输，和
- 本地传输（参看 org.jboss.netty.channel.local).

从一种传输方式切换到另外一种，通常只需要几行代码的变更，如选择另一种 ChannelFactory 的实现。

同样，你可以利用一个目前尚未编写的传输方式的优势，比如串口通讯的传输方式。再说一下，这只需要替换掉几行构造方法的代码。此外，核心 API 是高度可扩展的，你可以写你自己的传输方式。

2.3. 基于拦截者链模式的事件模型

定义恰当的、可扩展的事件模型对于事件驱动的应用程序而言，是必须的。Netty 就具有这样一个针对 I/O 的事件模型。你还可以实现你自己的事件类型。因为事件类型的层级结构很严格，每个类型都和其他类型有着严格的界定。因此，你完全可以在不破坏现有代码的情况下实现你自己的事件。这也是区别于其他框架的另一个标志。很多 NIO 框架没有或提供有限的事件模型。当需要添加一个新的定制的事件类型时，必须调整现有的代码。

一个 ChannelEvent 由 ChannelPipeline 中的多个 ChannelHandler 来处理。这里的 pipeline 实现了一个 Intercepting Filter 模式的高级形式，使得用户可以完全控制事件是如何被处理的、handler 之间如何相互操作。比如，你可以定义当一个数据从套接字中读过来的时候，如何做。

```
public class MyReadHandler implements SimpleChannelHandler {

public void messageReceived(ChannelHandlerContextctx, MessageEventevt) {

        Object message = evt.getMessage();

        // Do something with the received message.

        ...

        // And forward the event to the next handler.

ctx.sendUpstream(evt);

    }

}
```

你还可以定义当其他 handler 请求写操作的时候，如何做：

```
public class MyWriteHandler implements SimpleChannelHandler {

public void writeRequested(ChannelHandlerContextctx, MessageEventevt) {

        Object message = evt.getMessage();

        // Do something with the message to be written.

        ...



        // And forward the event to the next handler.
```

```
        ctx.sendDownstream(evt);
    }
}
```

关于事件模型的更多信息，请参看 API 文档 ChannelEvent 和 ChannelPipeline。

2.4. 为快速开发的高级组件

前面提到的核心组件的上面，已经可以实现所有类型的网络应用了。Netty 提供了一些提高开发速度的高级特性。

2.4.1. 编码框架

如在 Section 1.8, “ 用 POJO 替代 ChannelBuffer ”，把协议的编码和业务逻辑分离总是好的。但实际真要从零做起来做到这点，是有难度的：必须处理消息的拆解；一些协议是多层的（如位于其他低层协议之上）；一些是很难用一个状态机实现的。

因此，一个好的网络应用框架应该提供一个可扩展的、可复用的、可单元测试的、多层的编码框架，从而得到可维护的用户编码。

不论你在写的网络协议是简单还是复杂、二进制还是文本、任何形式的，Netty 在核心之上提供了很多基本的和高级的编码器来解决你会遇到的各种问题。

2.4.2. SSL / TLS 支持

和过去的阻塞式 I/O 不同，在 NIO 中支持 SSL 可不是容易的事，不能简单地封装一个流来对数据进行加密和解密，而是必须使用 javax.net.ssl.SSLEngine。SSLEngine 是一个和 SSL 一样复杂的状态机，必须管理所有可能的状态，包括密码组、加密密钥的协商（或再次协商）、证书交换和验证。不仅如此，SSLEngine 并非我们期望的那种严格的线程安全。

在 Netty 中，SslHandler 负责管理 SSLEngine 所有讨厌的细节和缺陷，留给开发者的只是配置 SslHandler 并将其添加到 ChannelPipeline。开发者可以很容易的实现诸如 StartTLS 的高级特性。

2.4.3. HTTP 实现

毫无疑问，HTTP 是最流行的互联网协议。当下已经有很多 HTTP 协议的实现了，如 Servlet 容器。那么为什么 Netty 在核心的上层还有 HTTP 呢？

Netty 的 HTTP 支持和现有的 HTTP 库是完全不同的。它允许你完全控制 HTTP 消息如何在底层进行交互。因为基本上它是 HTTP 编码器和 HTTP 消息类的组合，所以并不存在什么必须严格的限制，如强制的线程模型。也就是说，你可以写你自己的 HTTP 客户端或服务器端，让他们按你的需要来运行。你对线程模型、连接的生命周期、分段编码以及任何 HTTP 规范允许做的，有完全的控制。

得益于这种高度可定制的特点，你可以写一个高效率的 HTTP 服务器，如：

- 需要长连接的聊天服务器以及服务器推送技术。(如 Comet 和 WebSockets)
- 需要保持连接，直至整个媒体都传递完毕的流媒体服务器(如 2 小时的电影)
- 允许超大文件上传，但不带来内存压力的文件服务器(如，每个请求上传 1G 字节)
- 连接了几万个第三方异步 WEB 服务的、可扩充的混搭客户端

2.4.4. Google Protocol Buffer 集成

Google Protocol Buffers 对于快速开发高效的、随时间逐步完备的二进制协议而言，是一个理想的解决方案。利用 ProtobufEncoder 和 ProtobufDecoder，你可以把 Google Buffers Compiler（protoc）生成的消息类转换为 Netty 编码。请看一下'LocalTime' 例子，展示了用 sample protocol definition 构建一个高性能的二进制协议客户端和服务器是非常容易的。

2.5. 总结

本章中，我们从特点出发，涉及了 Netty 的整体框架。Netty 还具有一个强大的框架，由三个组件组成：buffer、channel、事件模型。所有的更高级的特点，都是建立在这三种组件之上的。如果理解了这三个组件是如何配合工作的，理解本章后面部分简要提及的更高级的特点就不应该很困难了。

你可能就整个框架的细节、各个特性如何工作等还有疑惑。如果真是这样，请告诉我们以提高这个文档。