

译 Netty In Action中文版 - 第一章：Netty介绍

目录(?)

[-]

1. 为什么使用Netty
 1. 不是所有的网络框架都是一样的
 2. Netty的功能非常丰富
2. 异步设计
 1. Callbacks回调
 2. Futures
3. Java中的Blocking和non-blocking IO对比
4. NIO的问题和Netty中是如何解决这些问题的
 1. 跨平台和兼容性问题
 2. 扩展ByteBuffer
 3. NIO对缓冲区的聚合和分散操作可能会操作内存泄露
 4. Squashing the famous epoll bug
5. Summary

本章介绍

- Netty介绍
- 为什么要使用non-blocking IO(NIO)
- 阻塞IO(blocking IO)和非阻塞IO(non-blocking IO)对比
- Java NIO的问题和在Netty中的解决方案

Netty是基于Java NIO的网络应用框架，如果你是Java网络方面的新手，那么本章将是你学习Java网络应用的开始；对于有经验的开发者来说，学习本章内容也是很好的复习。如果你熟悉NIO和NIO2，你可以随时跳过本章直接从第二章开始学习。在你的机器上运行第二章编写的Netty服务器和客户端。

Netty是一个NIO client-server(客户端服务器)框架，使用Netty可以快速开发网络应用，例如服务器和客户端协议。Netty提供了一种新的方式来使开发网络应用程序，这种新的方式使得它很容易使用和有很强的扩展性。Netty的内部实现时很复杂的，但是Netty提供了简单易用的api从网络处理代码中解耦业务逻辑。Netty是完全基于NIO实现的，所以整个Netty都是异步的。

网络应用程序通常需要有较高的可扩展性，无论是Netty还是其他的基于Java NIO的框架，都会提供可扩展性的解决方案。Netty中一个关键组成部分是它的异步特性，本章将讨论同步(阻塞)和异步(非阻塞)的IO来说明为什么使用异步代码来解决扩展性问题以及如何使用异步。

对于那些初学网络变成的读者，本章将帮助您对网络应用的理解，以及Netty是如何实现他们的。它说明了如何使用基本的Java网络API，探讨Java网络API的优点和缺点并阐述Netty是如何解决Java中的问题的，比如Eploo错误或内存泄露问题。

在本章的结尾，你会明白什么是**Netty**以及**Netty**提供了什么，你会理解**Java NIO**和异步处理机制，并通过本书的其他章节加强理解。

1.1 为什么使用Netty?

David John Wheeler说过“在计算机科学中的所有问题都可以通过间接的方法解决。”作为一个NIO client-server框架，Netty提供了这样的一个间接的解决方法。Netty提供了高层次的抽象来简化TCP和UDP服务器的编程，但是你仍然可以使用底层地API。

(David John Wheeler有一句名言“计算机科学中的任何问题都可以通过加上一层逻辑层来解决”，这个原则在计算机各技术领域被广泛应用)

1.1.1 不是所有的网络框架都是一样的

Netty的“quick and easy(高性能和简单易用)”并不意味着编写的程序的性能和可维护性会受到影响。从Netty中实现的协议如FTP, SMTP, HTTP, WebSocket, SPDY以及各种二进制和基于文本的传统协议中获得的经验导致Netty的创始人要非常小心它的设计。Netty成功的提供了易于开发, 高性能和高稳定性, 以及较强的扩展性。

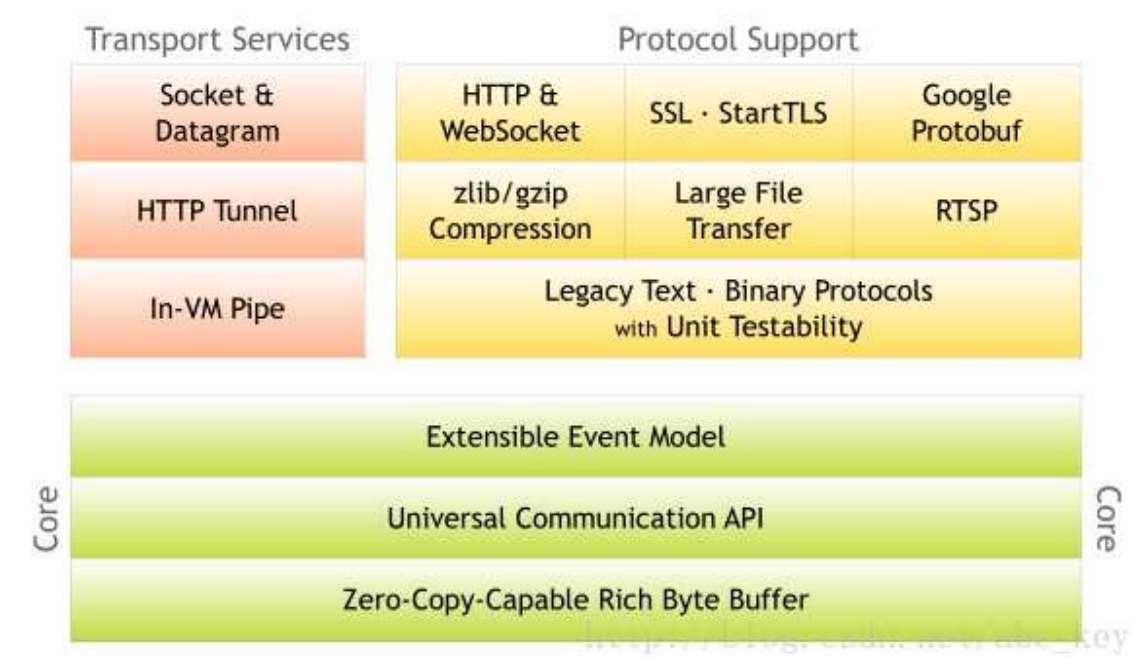
高调的公司和开源项目有RedHat, Twitter, Infinispan, and HornetQ, Vert.x, Finagle, Akka, Apache Cassandra, Elasticsearch, 以及其他人的使用有助于Netty的发展, Netty的一些特性也是这些项目的需要所致。多年来, Netty变的更广为人

知，它是Java网络的首选框架，在一些开源或非开源的项目中可以体现。并且，Netty在2011年获得Duke's Choice Award(Duke's Choice奖)。

此外，在2011年，Netty的创始人Trustion Lee离开RedHat后加入Twitter，在这一点上，Netty项目奖会成为一个独立的项目组织。RedHat和Twitter都使用Netty，所以它毫不奇怪。在撰写本书时RedHat和Twitter这两家公司是最大的贡献者。使用Netty的项目越来越多，Netty的用户群体和项目以及Netty社区都是非常活跃的。

1.1.2 Netty的功能非常丰富

通过本书可以学习Netty丰富的功能。下图是Netty框架的组成



Netty除了提供传输和协议，在其他各领域都有发展。Netty为开发者提供了一套完整的工具，看下面表格：

Development Area	Netty Features
Design(设计)	<ul style="list-style-type: none">• 各种传输类型，阻塞和非阻塞套接字统一的API• 使用灵活• 简单但功能强大的线程模型• 无连接的DatagramSocket支持• 链逻辑，易于重用
Ease of Use(易于使用)	<ul style="list-style-type: none">• 提供大量的文档和例子• 除了依赖jdk1.6+，没有额外的依赖关系。某些功能依赖jdk1.7+，其他特性可能有相关依赖，但都是可选的。
Performance(性能)	<ul style="list-style-type: none">• 比Java APIS更好的吞吐量和更低的延迟• 因为线程池和重用所有消耗较少的资源• 尽量减少不必要的内存拷贝
Robustness(鲁棒性)	<p>鲁棒性，可以理解为健壮性</p> <ul style="list-style-type: none">• 链接快或慢或超载不会导致更多的OutOfMemoryError• 在高速的网络程序中不会有不公平的read/write
Security(安全性)	<ul style="list-style-type: none">• 完整的SSL/TLS和StartTLS支持• 可以在如Applet或OSGI这些受限制的环境中运行
Community(社区)	<ul style="list-style-type: none">• 版本发布频繁• 社区活跃

除了列出的功能外，Netty为Java NIO中的bug和限制也提供了解决方案。我们需要深刻理解Netty的功能以及它的异步处理

机制和它的架构。NIO和Netty都大量使用了异步代码，并且封装的很好，我们无需了解底层的事件选择机制。下面我们来看看为什么需要异步APIS。

1.2 异步设计

整个Netty的API都是异步的，异步处理不是一个新的机制，这个机制出来已经有一些时间了。对网络应用来说，IO一般是性能的瓶颈，使用异步IO可以较大程度上提高程序性能，因为异步变的越来越重要。但是它是如何工作的呢？以及有哪些不同的模式可用呢？

异步处理提倡更有效的使用资源，它允许你创建一个任务，当有事件发生时将获得通知并等待事件完成。这样就不会阻塞，不管事件完成与否都会及时返回，资源利用率更高，程序可以利用剩余的资源做一些其他的事情。

本节将说明一起工作或实现异步API的两个最常用的方法，并讨论这些技术之间的差异。

1.2.1 Callbacks(回调)

回调一般是异步处理的一种技术。一个回调是被传递到并且执行完该方法。你可能认为这种模式来自JavaScript，在Javascript中，回调是它的核心。下面的代码显示了如何使用这种技术来获取数据。下面代码是一个简单的回调

[java]

```
01. package netty.in.action;
02.
03. public class Worker {
04.
05.     public void doWrk() {
06.         Fetcher fetcher = new MyFetcher(new Data(1, 0));
07.         fetcher.fetchData(new FetcherCallback() {
08.             @Override
09.             public void onError(Throwable cause) {
10.                 System.out.println("An error accour: " + cause.getMessage());
11.             }
12.
13.             @Override
14.             public void onData(Data data) {
15.                 System.out.println("Data received: " + data);
16.             }
17.         });
18.     }
19.
20.     public static void main(String[] args) {
21.         Worker w = new Worker();
22.         w.doWrk();
23.     }
24.
25. }
```

[java]

```
01. package netty.in.action;
02.
03. public interface Fetcher {
04.     void fetchData(FetcherCallback callback);
05. }
```

[java]

```
01. package netty.in.action;
02.
03. public class MyFetcher implements Fetcher {
04.
05.     final Data data;
06.
07.     public MyFetcher(Data data){
08.         this.data = data;
09.     }
10.
11.     @Override
12.     public void fetchData(FetcherCallback callback) {
13.         try {
14.             callback.onData(data);
15.         } catch (Exception e) {
16.             callback.onError(e);
17.         }
18.     }
19. }
```

```
18.     }
19.
20. }
```

```
[java]
01. package netty.in.action;
02.
03. public interface FetcherCallback {
04.     void onData(Data data) throws Exception;
05.     void onError(Throwable cause);
06. }
```

```
[java]
01. package netty.in.action;
02.
03. public class Data {
04.
05.     private int n;
06.     private int m;
07.
08.     public Data(int n,int m){
09.         this.n = n;
10.         this.m = m;
11.     }
12.
13.     @Override
14.     public String toString() {
15.         int r = n/m;
16.         return n + "/" + m + " = " + r;
17.     }
18. }
```

上面的例子只是一个简单的模拟回调，要明白其所表达的含义。`Fetcher.fetchData()`方法需传递一个`FetcherCallback`类型的参数，当获得数据或发生错误时被回调。对于每种情况都提供了同意的方法：

- `FetcherCallback.onData()`，将接收数据时被调用
- `FetcherCallback.onError()`，发生错误时被调用

因为可以将这些方法的执行从"caller"线程移动到其他的线程执行；但也不会保证`FetcherCallback`的每个方法都会被执行。回调过程有个问题就是当你使用链式调用很多不同的方法会导致线性代码；有些人认为这种链式调用方法会导致代码难以阅读，但是我认为这是一种风格和习惯问题。例如，基于Javascript的Node.js越来越受欢迎，它使用了大量的回调，许多人都认为它的这种方式利于阅读和编写。

1.2.2 Futures

第二种技术是使用Futures。Futures是一个抽象的概念，它表示一个值，该值可能在某一点变得可用。一个Future要么获得计算完的结果，要么获得计算失败后的异常。Java在java.util.concurrent包中附带了Future接口，它使用Executor异步执行。例如下面的代码，每传递一个Runnable对象到ExecutorService.submit()方法就会得到一个回调的Future，你能使用它检测是否执行完成。

```
[java]
01. package netty.in.action;
02.
03. import java.util.concurrent.Callable;
04. import java.util.concurrent.ExecutorService;
05. import java.util.concurrent.Executors;
06. import java.util.concurrent.Future;
07.
08. public class FutureExample {
09.
10.     public static void main(String[] args) throws Exception {
11.         ExecutorService executor = Executors.newCachedThreadPool();
12.         Runnable task1 = new Runnable() {
13.             @Override
14.             public void run() {
15.                 //do something
16.                 System.out.println("i am task1....");
```

```
17.         }
18.     };
19.     Callable<Integer> task2 = new Callable<Integer>() {
20.         @Override
21.         public Integer call() throws Exception {
22.             //do something
23.             return new Integer(100);
24.         }
25.     };
26.     Future<?> f1 = executor.submit(task1);
27.     Future<Integer> f2 = executor.submit(task2);
28.     System.out.println("task1 is completed? " + f1.isDone());
29.     System.out.println("task2 is completed? " + f2.isDone());
30.     //waiting task1 completed
31.     while(f1.isDone()){
32.         System.out.println("task1 completed.");
33.         break;
34.     }
35.     //waiting task2 completed
36.     while(f2.isDone()){
37.         System.out.println("return value by task2: " + f2.get());
38.         break;
39.     }
40. }
41.
42. }
```

有时候使用Future感觉很丑陋，因为你需要间隔检查Future是否已完成，而使用回调会直接收到返回通知。看完这两个常用的异步执行技术后，你可能想知道使用哪个最好？这里没有明确的答案。事实上，Netty两者都使用，提供两全其美的方案。下一节将在JVM上首先使用阻塞，然后再使用NIO和NIO2写一个网络程序。这些是本书后续章节必不可少的基础知识，如果你熟悉Java网络AIPs，你可以快速翻阅即可。

1.3 Java中的Blocking和non-blocking IO对比

本节主要讲解Java的IO和NIO的差异，这里不过多赘述，网络已有很多相关文章。

1.4 NIO的问题和Netty中是如何解决这些问题的

本节中将介绍Netty是如何解决NIO中的一些问题和限制。Java的NIO相对老的IO APIs有着非常大的进步，但是使用NIO是受限制的。这些问题往往是设计的问题，有些是缺陷知道的。

1.4.1 跨平台和兼容性问题

NIO是一个比较底层的APIs，它依赖于操作系统的IO APIs。Java实现了统一的接口来操作IO，其在所有操作系统中的工作行为是一样的，这是很伟大的。使用NIO会经常发现代码在Linux上正常运行，但在Windows上就会出现問題。我建议你如果使用NIO编写程序，就应该在所有的操作系统上进行测试来支持，使程序可以在任何操作系统上正常运行；即使在所有的Linux系统上都测试通过了，也要在其他的操作系统上进行测试；你若不验证，以后就可能会出问题。

NIO2看起来很理想，但是NIO2只支持Jdk1.7+，若你的程序在Java1.6上运行，则无法使用NIO2。另外，Java7的NIO2中没有提供DatagramSocket的支持，所以NIO2只支持TCP程序，不支持UDP程序。

Netty提供一个统一的接口，同一语义无论在Java6还是Java7的环境下都是可以运行的，开发者无需关心底层APIs就可以轻松实现相关功能。

1.4.2 扩展ByteBuffer

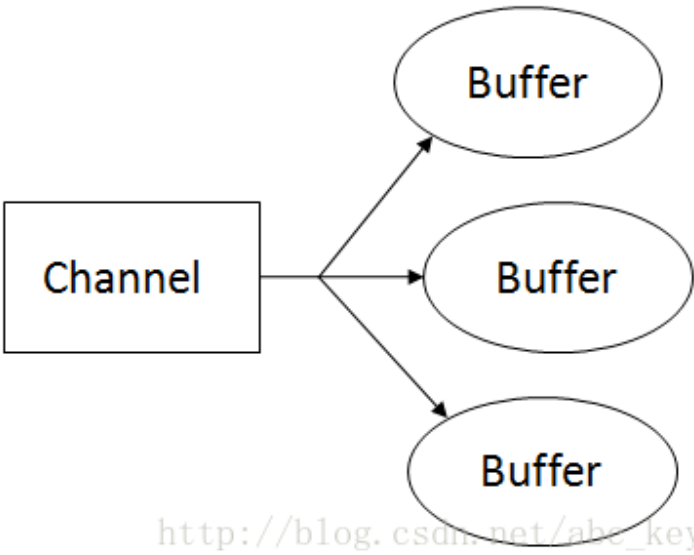
ByteBuffer是一个数据容器，但是可惜的是JDK没有开发ByteBuffer实现的源码；ByteBuffer允许包装一个byte[]来获得一个实例，如果你希望尽量减少内存拷贝，那么这种方式是非常有用的。若果你想将ByteBuffer重新实现，那么不要浪费你的时间了，ByteBuffer的构造函数是私有的，所以它不能被扩展。Netty提供了自己的ByteBuffer实现，Netty通过一些简单的APIs对ByteBuffer进行构造、使用和操作，以此来解决NIO中的一些限制。

1.4.3 NIO对缓冲区的聚合和分散操作可能会操作内存泄露

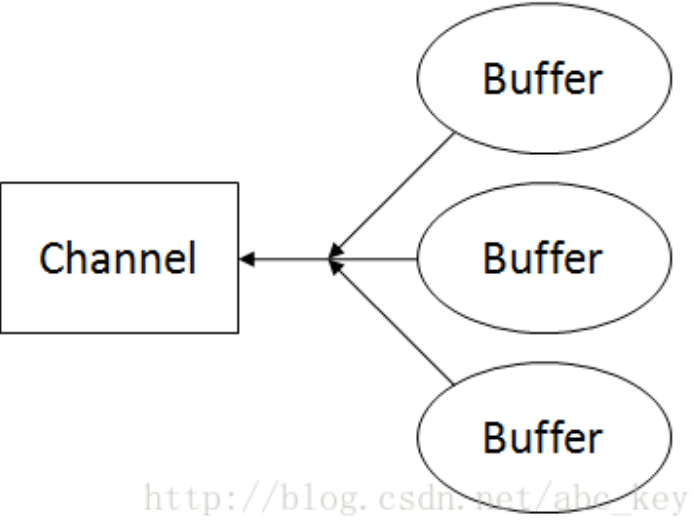
很多Channel的实现支持Gather和Scatter。这个功能允许从多个ByteBuffer中读入或写入到过个ByteBuffer，这样做可以提供性能。操作系统底层知道如何处理这些被写入/读出，并且能以最有效的方式处理。如果要分割的数据再多个不同的ByteBuffer中，使用Gather/Scatter是比较好的方式。

例如，你可能希望header在一个ByteBuffer中，而body在另外的ByteBuffer中；

下图显示的是Scatter(分散)，将ScatteringByteBuffer中的数据分散读取到多个ByteBuffer中：



下图显示的是Gather(聚合)，将多个ByteBuffer的数据写入到GatheringByteChannel:



可惜Gather/Scatter功能会导致内存泄露，知道Java7才解决内存泄露问题。使用这个功能必须小心编码和Java版本。

1.4.4 Squashing the famous epoll bug

压碎著名的epoll缺陷。

On Linux-like OSs the selector makes use of the epoll- IO event notification facility. This is a high-performance technique in which the OS works asynchronously with the networking stack.Unfortunately, even today the "famous" epoll- bug can lead to an "invalid" state in the selector, resulting in 100% CPU-usage and spinning. The only way to recover is to recycle the old selector and transfer the previously registered Channel instances to the newly created Selector.

Linux-like OSs的选择器使用的是epoll-IO事件通知工具。这是一个在操作系统以异步方式工作的网络stack.Unfortunately，即使是现在，著名的epoll-bug也可能会导致无效的状态的选择和100%的CPU利用率。要解决epoll-bug的唯一方法是回收旧的选择器，将先前注册的通道实例转移到新创建的选择器上。

What happens here is that the Selector.select() method stops to block and returns immediately-even if there are no selected SelectionKeys present. This is against the contract, which is in the Javadocs of the Selector.select() method:Selector.select() must not unblock if nothing is selected.

这里发生的是，不管有没有已选择的SelectionKey，Selector.select()方法总是不会阻塞并且会立刻返回。这违反了Javadoc中对Selector.select()方法的描述，Javadoc中的描述：Selector.select() must not unblock if nothing is selected. (Selector.select()方法若未选中任何事件将会阻塞。)

The range of solutions to this epoll- problem is limited, but Netty attempts to automatically detect and prevent it. The following listing is an example of the epoll- bug.

NIO中对epoll问题的解决方案是有限制的，Netty提供了更好的解决方案。下面是epoll-bug的一个例子：

```
...
while (true) {
int selected = selector.select();
Set<SelectedKeys> readyKeys = selector.selectedKeys();
Iterator iterator = readyKeys.iterator();
while (iterator.hasNext()) {
...
...
}
```

```
}  
}  
...
```

The effect of this code is that the while loop eats CPU:
这段代码的作用是while循环消耗CPU:

```
...  
while (true) {  
}  
...
```

The value will never be false, and the code keeps your CPU spinning and eats resources. This can have some undesirable side effects as it can consume all of your CPU, preventing any other CPU-bound work.

该值将永远是假的，代码将持续消耗你的CPU资源。这会有一些副作用，因为CPU消耗完了就无法再去做其他任何的工作。

These are only a few of the possible problems you may see while using non-blocking IO. Unfortunately, even after years of development in this area, issues still need to be resolved; thankfully, Netty addresses them for you.

这些仅仅是在使用NIO时可能会出现的一些问题。不幸的是，虽然在这个领域发展了多年，问题依然存在；幸运的是，Netty 给了你解决方案。

1.5 Summary

This chapter provided an overview of Netty's features, design and benefits. I discussed the difference between blocking and non-blocking processing to give you a fundamental understanding of the reasons to use a non-blocking framework. You learned how to use the JDK API to write network code in both blocking and non-blocking modes. This included the new non-blocking API, which comes with JDK 7. After seeing the NIO APIs in action, it was also important to understand some of the known issues that you may run into. In fact, this is why so many people use Netty: to take care of workarounds and other JVM quirks. In the next chapter, you'll learn the basics of the Netty API and programming model, and, finally, use Netty to write some useful code.

目录(?)	[+]
1. 设置开发环境	
2. Netty客户端和服务端概述	
3. 编写一个应答服务器	
1. 启动服务器	
2. 实现服务器业务逻辑	
3. 捕获异常	
4. 编写应答程序的客户端	
1. 引导客户端	
2. 实现客户端的业务逻辑	
5. 编译和运行echo应答程序客户端和服务端	
6. 总结	

本章介绍

- 获取Netty4最新版本
- 设置运行环境来构建和运行netty程序
- 创建一个基于Netty的服务器和客户端
- 拦截和处理异常
- 编写和运行Netty服务器和客户端

本章将简单介绍Netty的核心概念，这个狠心概念就是学习Netty是如何拦截和处理异常，对于刚开始学习netty的读者，利用netty的异常拦截机制来调试程序问题很有帮助。本章还会介绍其他一些核心概念，如服务器和客户端的启动以及分离通道的处理程序。本章学习一些基础以便后面章节的深入学习。本章中将编写一个基于netty的服务器和客户端来互相通信，我们首先来设置netty的开发环境。

2.1 设置开发环境

设置开发环境的步骤如下：

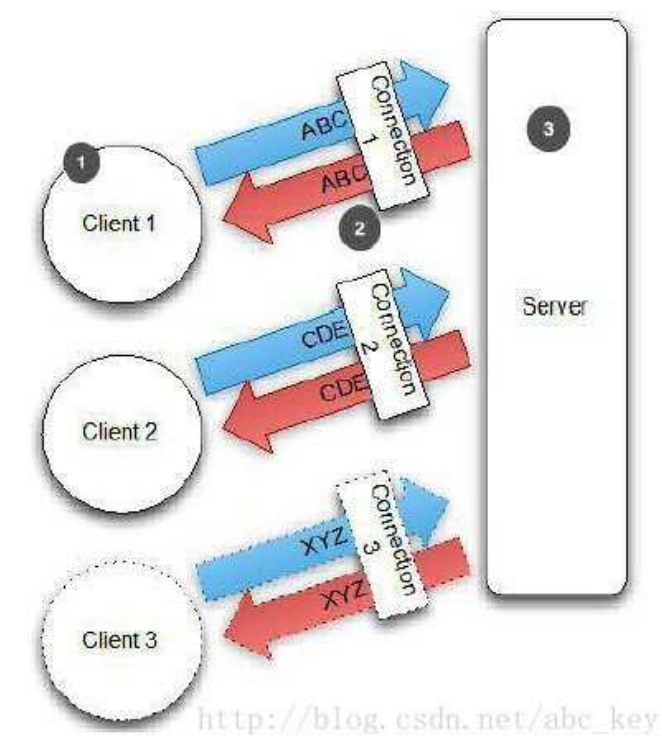
- 安装JDK7，下载地址<http://www.oracle.com/technetwork/java/javase/archive-139210.html>
- 下载netty包，下载地址<http://netty.io/>
- 安装Eclipse

《Netty In Action》中描述的比较多，没啥用，这里就不多说了。本系列博客将使用Netty4，需要JDK1.7+

2.2 Netty客户端和服务端概述

本节将引导你构建一个完整的Netty服务器和客户端。一般情况下，你可能只关心编写服务器，如一个http服务器的客户端是浏览器。然后在这个例子中，你若同时实现了服务器和客户端，你将会对他们的原理更加清晰。

一个Netty程序的工作图如下



1. 客户端连接到服务器
2. 建立连接后，发送或接收数据
3. 服务器处理所有的客户端连接

从上图中可以看出，服务器会写数据到客户端并且处理多个客户端的并发连接。从理论上来说，限制程序性能的因素只有系统资源和JVM。为了方便理解，这里举了个生活例子，在山谷或高山上大声喊，你会听见回声，回声是山返回的；在这个例子中，你是客户端，山是服务器。喊的行为就类似于一个Netty客户端将数据发送到服务器，听到回声就类似于服务器将相同的数据返回给你，你离开山谷就断开了连接，但是你可以返回进行重连

服务器并且可以发送更多的数据。

虽然将相同的数据返回给客户端不是一个典型的例子，但是客户端和服务端之间数据的来来回回的传输和这个例子是一样的。本章的例子会证明这一点，它们会越来越复杂。

接下来的几节将带着你完成基于Netty的客户端和服务器的应答程序。

2.3 编写一个应答服务器

写一个Netty服务器主要由两部分组成：

- 配置服务器功能，如线程、端口
- 实现服务器处理程序，它包含业务逻辑，决定当有一个请求连接或接收数据时该做什么

2.3.1 启动服务器

通过创建ServerBootstrap对象来启动服务器，然后配置这个对象的相关选项，如端口、线程模式、事件循环，并且添加逻辑处理程序用来处理业务逻辑(下面是个简单的应答服务器例子)

```
[java]
01. package netty.example;
02.
03. import io.netty.bootstrap.ServerBootstrap;
04. import io.netty.channel.Channel;
05. import io.netty.channel.ChannelFuture;
06. import io.netty.channel.ChannelInitializer;
07. import io.netty.channel.EventLoopGroup;
08. import io.netty.channel.nio.NioEventLoopGroup;
09. import io.netty.channel.socket.nio.NioServerSocketChannel;
10.
11. public class EchoServer {
12.
13.     private final int port;
14.
15.     public EchoServer(int port) {
16.         this.port = port;
17.     }
18.
19.     public void start() throws Exception {
20.         EventLoopGroup group = new NioEventLoopGroup();
21.         try {
22.             //create ServerBootstrap instance
23.             ServerBootstrap b = new ServerBootstrap();
24.             //Specifies NIO transport, local socket address
25.             //Adds handler to channel pipeline
26.             b.group(group).channel(NioServerSocketChannel.class).localAddress(port)
27.                 .childHandler(new ChannelInitializer<Channel>() {
28.                     @Override
29.                     protected void initChannel(Channel ch) throws Exception {
30.                         ch.pipeline().addLast(new EchoServerHandler());
31.                     }
32.                 });
33.             //Binds server, waits for server to close, and releases resources
34.             ChannelFuture f = b.bind().sync();
35.             System.out.println(EchoServer.class.getName() + "started and listen on " + f.channel().localAddress());
36.             f.channel().closeFuture().sync();
37.         } finally {
38.             group.shutdownGracefully().sync();
39.         }
40.     }
41.
42.     public static void main(String[] args) throws Exception {
43.         new EchoServer(65535).start();
44.     }
45.
46. }
```

从上面这个简单的服务器例子可以看出，启动服务器应先创建一个ServerBootstrap对象，因为使用NIO，所以指定NioEventLoopGroup来接受和处理新连接，指定通道类型为NioServerSocketChannel，设置InetSocketAddress让服务器监听某个端口已等待客户端连接。

接下来，调用childHandler放来指定连接后调用的ChannelHandler，这个方法传ChannelInitializer类型的参数，ChannelInitializer是个抽象类，所以需要实现initChannel方法，这个方法就是用来设置ChannelHandler。

最后绑定服务器等待直到绑定完成，调用sync()方法会阻塞直到服务器完成绑定，然后服务器等待通道关闭，因为使用sync()，所以关闭操作也会被阻塞。现在你可以关闭EventLoopGroup和释放所有资源，包括创建的线程。

这个例子中使用NIO，因为它是目前最常用的传输方式，你可能会使用NIO很长时间，但是你可以选择不同的传输实现。例如，这个例子使用OIO方式传输，你需要指定OioServerSocketChannel。Netty框架中实现了多重传输方式，将再后面讲述。

本小节重点内容：

- 创建ServerBootstrap实例来引导绑定和启动服务器
- 创建NioEventLoopGroup对象来处理事件，如接受新连接、接收数据、写数据等等

- 指定InetSocketAddress，服务器监听此端口
- 设置childHandler执行所有的连接请求
- 都设置完毕了，最后调用ServerBootstrap.bind() 方法来绑定服务器

2.3.2 实现服务器业务逻辑

Netty使用futures和回调概念，它的设计允许你处理不同的事件类型，更详细的介绍将再后面章节讲述，但是我们可以接收数据。你的channel handler必须继承ChannelInboundHandlerAdapter并且重写channelRead方法，这个方法在任何时候都会被调用来接收数据，在这个例子中接收的是字节。

下面是handler的实现，其实现的功能是将客户端发给服务器的数据返回给客户端：

```
[java]
01. package netty.example;
02.
03. import io.netty.buffer.Unpooled;
04. import io.netty.channel.ChannelFutureListener;
05. import io.netty.channel.ChannelHandlerContext;
06. import io.netty.channel.ChannelInboundHandlerAdapter;
07.
08. public class EchoServerHandler extends ChannelInboundHandlerAdapter {
09.
10.     @Override
11.     public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {
12.         System.out.println("Server received: " + msg);
13.         ctx.write(msg);
14.     }
15.
16.     @Override
17.     public void channelReadComplete(ChannelHandlerContext ctx) throws Exception {
18.         ctx.writeAndFlush(Unpooled.EMPTY_BUFFER).addListener(ChannelFutureListener.CLOSE);
19.     }
20.
21.     @Override
22.     public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws Exception {
23.         cause.printStackTrace();
24.         ctx.close();
25.     }
26.
27. }
```

Netty使用多个Channel Handler来达到对事件处理的分离，因为可以很容的添加、更新、删除业务逻辑处理handler。Handler很简单，它的每个方法都可以被重写，它的所有的方法中只有channelRead方法是必须要重写的。

2.3.3 捕获异常

重写ChannelHandler的exceptionCaught方法可以捕获服务器的异常，比如客户端连接服务器后强制关闭，服务器会抛出"客户端主机强制关闭错误"，通过重写exceptionCaught方法就可以处理异常，比如发生后关闭ChannelHandlerContext。

2.4 编写应答程序的客户端

服务器写好了，现在来写一个客户端连接服务器。应答程序的客户端包括以下几步：

- 连接服务器
- 写数据到服务器
- 等待接受服务器返回相同的数据
- 关闭连接

2.4.1 引导客户端

引导客户端启动和引导服务器很类似，客户端需同时指定host和port来告诉客户端连接哪个服务器。看下面代码：

```
[java]
01. package netty.example;
02.
03. import io.netty.bootstrap.Bootstrap;
04. import io.netty.channel.ChannelFuture;
05. import io.netty.channel.ChannelInitializer;
06. import io.netty.channel.EventLoopGroup;
07. import io.netty.channel.nio.NioEventLoopGroup;
08. import io.netty.channel.socket.SocketChannel;
09. import io.netty.channel.socket.nio.NioSocketChannel;
10. import io.netty.example.echo.EchoClientHandler;
11.
12. import java.net.InetSocketAddress;
13.
14. public class EchoClient {
15.
16.     private final String host;
17.     private final int port;
18.
19.     public EchoClient(String host, int port) {
20.         this.host = host;
```

```
21.         this.port = port;
22.     }
23.
24.     public void start() throws Exception {
25.         EventLoopGroup group = new NioEventLoopGroup();
26.         try {
27.             Bootstrap b = new Bootstrap();
28.             b.group(group).channel(NioSocketChannel.class).remoteAddress(new InetSocketAddress(host, port))
29.                 .handler(new ChannelInitializer<SocketChannel>() {
30.                     @Override
31.                     protected void initChannel(SocketChannel ch) throws Exception {
32.                         ch.pipeline().addLast(new EchoClientHandler());
33.                     }
34.                 });
35.             ChannelFuture f = b.connect().sync();
36.             f.channel().closeFuture().sync();
37.         } finally {
38.             group.shutdownGracefully().sync();
39.         }
40.     }
41.
42.     public static void main(String[] args) throws Exception {
43.         new EchoClient("localhost", 20000).start();
44.     }
45. }
```

创建启动一个客户端包含下面几步：

- 创建Bootstrap对象用来引导启动客户端
- 创建EventLoopGroup对象并设置到Bootstrap中，EventLoopGroup可以理解为是一个线程池，这个线程池用来处理连接、接受数据、发送数据
- 创建InetSocketAddress并设置到Bootstrap中，InetSocketAddress是指定连接的服务器地址
- 添加一个ChannelHandler，客户端成功连接服务器后就会被执行
- 调用Bootstrap.connect()来连接服务器
- 最后关闭EventLoopGroup来释放资源

2.4.2 实现客户端的业务逻辑

客户端的业务逻辑的实现依然很简单，更复杂的用法将在后面章节详细介绍。和编写服务器的ChannelHandler一样，在这里将自定义一个继承SimpleChannelInboundHandler的ChannelHandler来处理业务；通过重写父类的三个方法来处理感兴趣的事件：

- channelActive(): 客户端连接服务器后被调用
- channelRead0(): 从服务器接收到数据后调用
- exceptionCaught(): 发生异常时被调用

实现代码如下

```
[java]
01. package netty.example;
02.
03. import io.netty.buffer.ByteBuf;
04. import io.netty.buffer.ByteBufUtil;
05. import io.netty.buffer.Unpooled;
06. import io.netty.channel.ChannelHandlerContext;
07. import io.netty.channel.SimpleChannelInboundHandler;
08. import io.netty.util.CharsetUtil;
09.
10. public class EchoClientHandler extends SimpleChannelInboundHandler<ByteBuf> {
11.
12.     @Override
13.     public void channelActive(ChannelHandlerContext ctx) throws Exception {
14.         ctx.write(Unpooled.copiedBuffer("Netty rocks!", CharsetUtil.UTF_8));
15.     }
16.
17.     @Override
18.     protected void channelRead0(ChannelHandlerContext ctx, ByteBuf msg) throws Exception {
19.         System.out.println("Client received: " + ByteBufUtil.hexDump(msg.readBytes(msg.readableBytes())));
20.     }
21.
22.     @Override
23.     public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws Exception {
24.         cause.printStackTrace();
25.         ctx.close();
26.     }
27. }
```

可能你会问为什么在这里使用的是SimpleChannelInboundHandler而不使用ChannelInboundHandlerAdapter？主要原因是ChannelInboundHandlerAdapter在处理完消息后需要负责释放资源。在这里将调用ByteBuf.release()来释放资源。SimpleChannelInboundHandler会在完成channelRead0后释放消息，这是通过Netty处理所有消息的ChannelHandler实现了ReferenceCounted接口达到的。

为什么在服务器中不使用SimpleChannelInboundHandler呢？因为服务器要返回相同的消息给客户端，在服务器执行完成写操作之前不能释放调

用读取到的消息，因为写操作是异步的，一旦写操作完成后，Netty中会自动释放消息。

客户端的编写完了，下面让我们来测试一下

2.5 编译和运行echo(应答)程序客户端和服务端

注意，netty4需要jdk1.7+。

本人测试，可以正常运行。

2.6 总结

本章介绍了如何编写一个简单的基于Netty的服务器和客户端并进行通信发送数据。介绍了如何创建服务器和客户端以及Netty的异常处理机制。

目录(?)

[1]

1. Netty Crash Course

2. ChannelsEvents and InputOutputIO

3. 什么是Bootstrap为什么使用它

4. Channel Handlers and Data Flow通道处理和数据流

5. 编码器解码器和业务逻辑细看Handlers

1. Encoders编码器 decoders解码器

2. 业务逻辑Domain logic

在这一章我们将讨论Netty的10个核心类，清楚了解他们的结构对使用Netty很有用。可能有一些不会再工作中用到，但是也有一些很常用也很核心，你会遇到。

- Bootstrap or ServerBootstrap
- EventLoop
- EventLoopGroup
- ChannelPipeline
- Channel
- Future or ChannelFuture
- ChannelInitializer
- ChannelHandler

本节的目的就是介绍以上这些概念，帮助你了解它们的用法。

3.1 Netty Crash Course

在我们开始之前，如果你了解Netty程序的一般结构和大致用法(客户端和服务端都有一个类似的结构)会更好。

一个Netty程序开始于Bootstrap类，Bootstrap类是Netty提供的一个可以通过简单配置来设置或"引导"程序的一个很重要的类。Netty中设计了Handlers来处理特定的"event"和设置Netty中的事件，从而来处理多个协议和数据。事件可以描述成一个非常通用的方法，因为你可以自定义一个handler,用来将Object转成byte[]或将byte[]转成Object；也可以定义个handler处理抛出的异常。

你会经常编写一个实现ChannelInboundHandler的类，ChannelInboundHandler是用来接收消息，当有消息过来时，你可以决定如何处理。当程序需要返回消息时可以在ChannelInboundHandler里write/flush数据。可以认为应用程序的业务逻辑都是在ChannelInboundHandler中处理的，业务罗的生命周期在ChannelInboundHandler中。

Netty连接客户端端或绑定服务器需要知道如何发送或接收消息，这是通过不同类型的handlers来做的，多个Handlers是怎么配置的？Netty提供了ChannelInitializer类用来配置Handlers。ChannelInitializer是通过ChannelPipeline来添加ChannelHandler的，如发送和接收消息，这些Handlers将确定发的是什么消息。ChannelInitializer自身也是一个ChannelHandler，在添加完其他的handlers之后会自动从ChannelPipeline中删除自己。

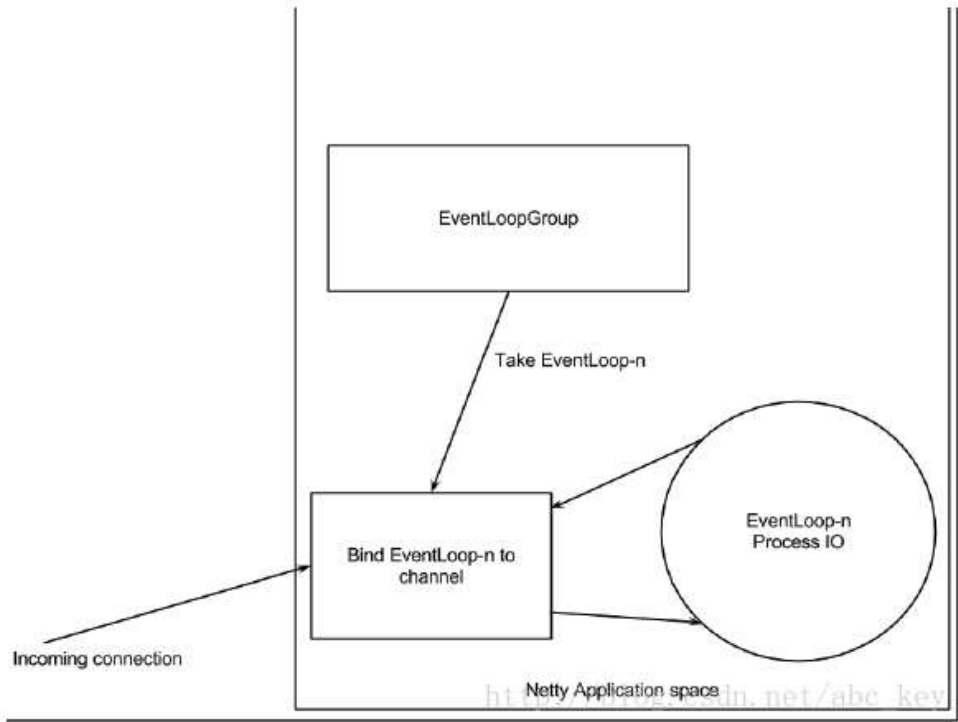
所有的Netty程序都是基于ChannelPipeline。ChannelPipeline和EventLoop和EventLoopGroup密切相关，因为它们三个都和事件处理相关，所以这就是为什么它们处理IO的工作由EventLoop管理的原因。

Netty中所有的IO操作都是异步执行的，例如你连接一个主机默认是异步完成的；写入/发送消息也是同样是异步。也就是说操作不会直接执行，而是会等一会执行，因为你不知道返回的操作结果是成功还是失败，但是需要有检查是否成功的方法或者是注册监听来通知；Netty使用Futures和ChannelFutures来达到这种目的。Future注册一个监听，当操作成功或失败时会通知。ChannelFuture封装的是一个操作的相关信息，操作被执行时会立刻返回ChannelFuture。

3.2 Channels,Events and Input/Output(IO)

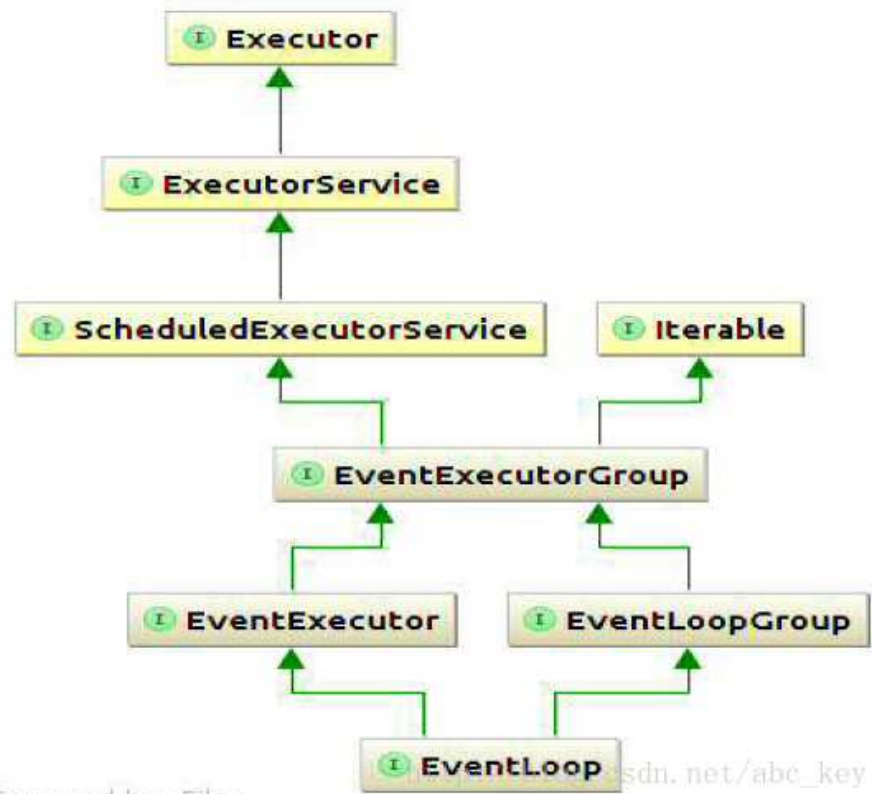
Netty是一个非阻塞、事件驱动的网络框架。Netty实际上是使用多线程处理IO事件，对于熟悉多线程编程的读者可能会需要同步代码。这样的方式不好，因为同步会影响程序的性能，Netty的设计保证程序处理事件不会有同步。

下图显示一个EventLoopGroup和一个Channel关联一个单一的EventLoop，Netty中的EventLoopGroup包含一个或多个EventLoop，而EventLoop就是一个Channel执行实际工作的线程。EventLoop总是绑定一个单一的线程，在其生命周期内不会改变。



当注册一个Channel后，Netty将这个Channel绑定到一个EventLoop，在Channel的生命周期内总是被绑定到一个EventLoop。在Netty IO操作中，你的程序不需要同步，因为一个指定通道的所有IO始终由同一个线程来执行。

为了帮助理解，下图显示了EventLoop和EventLoopGroup的关系：



EventLoop和EventLoopGroup的关联不是直观的，因为我们说过EventLoopGroup包含一个或多个EventLoop，但是上面的图显示EventLoop是一个EventLoopGroup，这意味着你可以只使用一个特定的EventLoop。

3.3 什么是Bootstrap?为什么使用它？

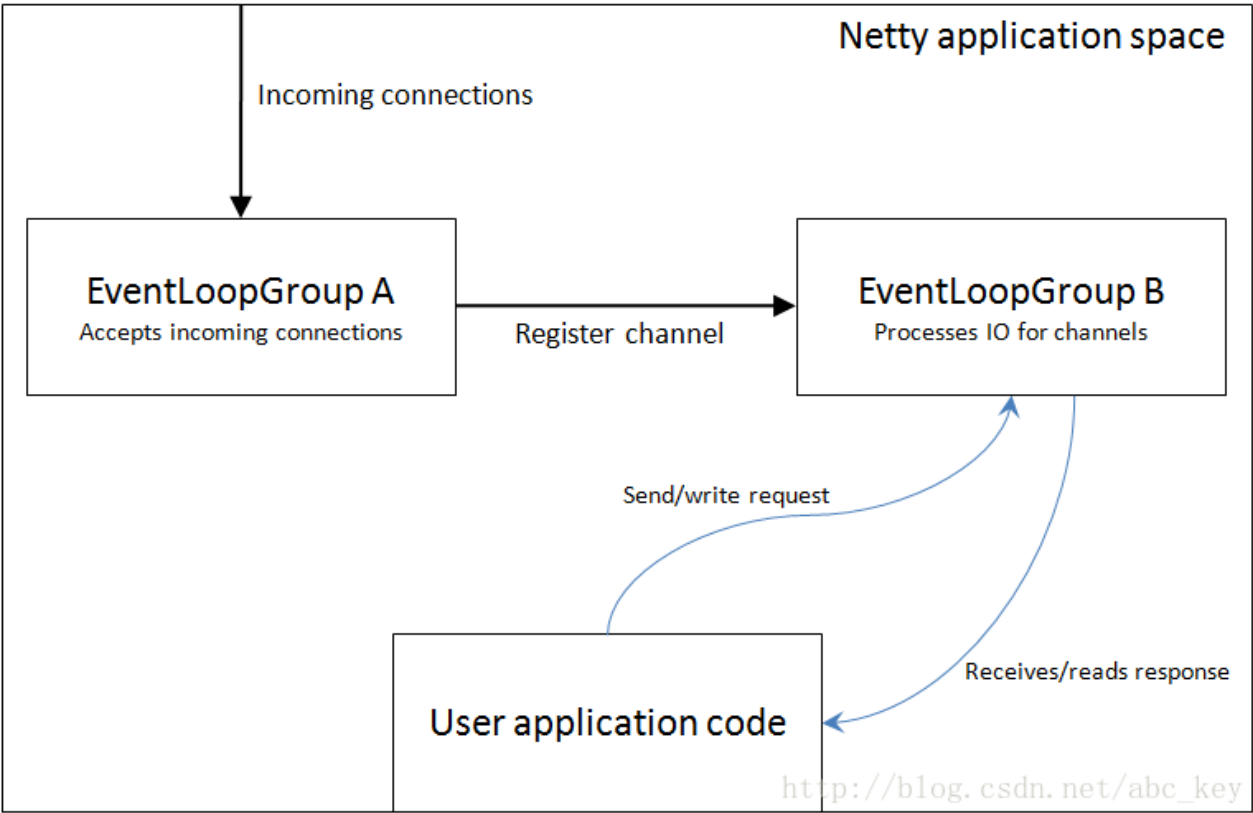
“引导”是Netty中配置程序的过程，当你需要连接客户端或服务器绑定指定端口时需要使用bootstrap。如前面所述，“引导”有两种类型，一种是用于客户端的Bootstrap(也适用于DatagramChannel)，一种是用于服务端的ServerBootstrap。不管程序使用哪种协议，无论是创建一个客户端还是服务器都需要使用“引导”。

两种bootstraps之间有一些相似之处，其实他们有很多相似之处，也有一些不同。Bootstrap和ServerBootstrap之间的差异：

- Bootstrap用来连接远程主机，有1个EventLoopGroup
- ServerBootstrap用来绑定本地端口，有2个EventLoopGroup

事件组(Groups)，传输(transports)和处理程序(handlers)分别在本章后面讲述，我们在这里只讨论两种"引导"的差异(Bootstrap和ServerBootstrap)。第一个差异很明显，“ServerBootstrap”监听在服务器监听一个端口轮询客户端的“Bootstrap”或DatagramChannel是否连接服务器。通常需要调用“Bootstrap”类的connect()方法，但是也可以先调用bind()再调用connect()进行连接，之后使用的Channel包含在bind()返回的ChannelFuture中。

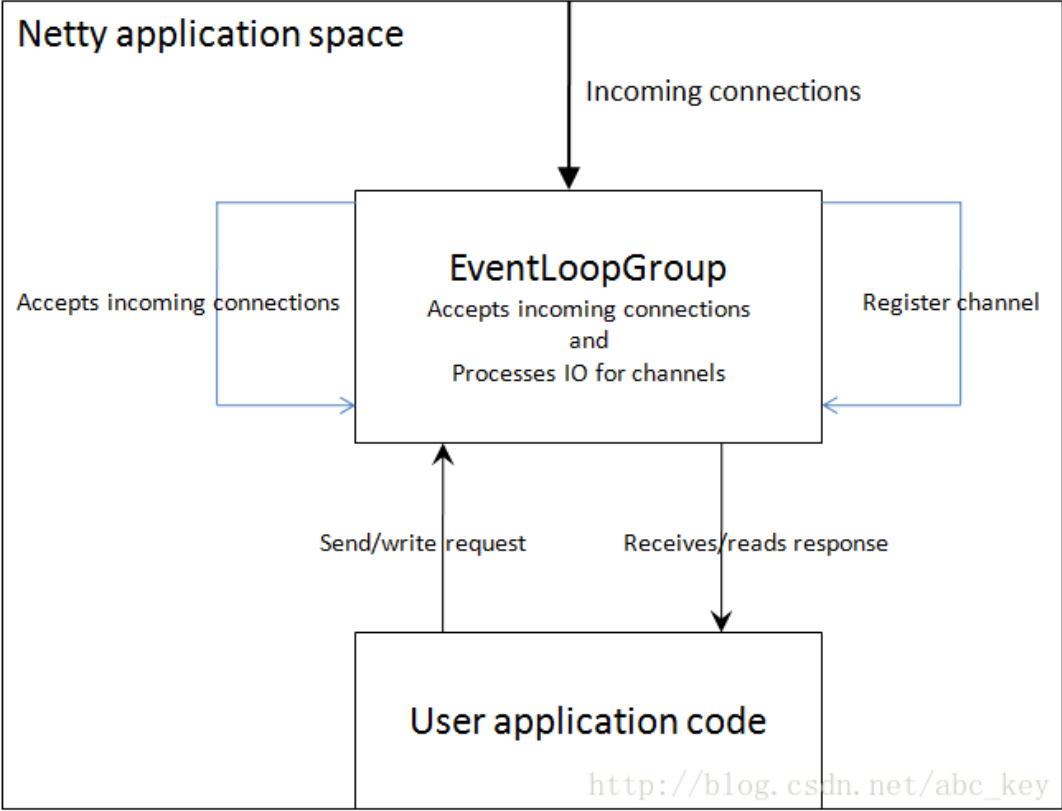
第二个差别也许是最重要的。客户端bootstraps/applications使用一个单例EventLoopGroup，而ServerBootstrap使用2个EventLoopGroup(实际上使用的是相同的实例)，它可能不是显而易见的，但是它是个好的方案。一个ServerBootstrap可以认为有2个channels组，第一组包含一个单例ServerChannel，代表持有一个绑定了本地端口的socket；第二组包含所有的Channel，代表服务器已接受了的连接。下图形象的描述了这种情况：



上图中，EventLoopGroup A唯一的目的是接受连接然后交给EventLoopGroup B。Netty可以使用两个不同的Group，因为服务器程序需要接受很多客户端连接的情况下，一个EventLoopGroup将是程序性能的瓶颈，因为事件循环忙于处理连接请求，没有多余的资源和空闲来处理业务逻辑，最后的结果会是很多连接请求超时。若有两EventLoops，即使在高负载下，所有的连接也都会被接受，因为EventLoops接受连接不会和哪些已经连接了的处理共享资源。

EventLoopGroup和EventLoop是什么关系？EventLoopGroup可以包含很多个EventLoop，每个Channel绑定一个EventLoop不会被改变，因为EventLoopGroup包含少量的EventLoop的Channels，很多Channel会共享同一个EventLoop。这意味着在一个Channel保持EventLoop繁忙会禁止其他Channel绑定到相同的EventLoop。我们可以理解为EventLoop是一个事件循环线程，而EventLoopGroup是一个事件循环集合。

如果你决定两次使用相同的EventLoopGroup实例配置Netty服务器，下图显示了它是如何改变的：



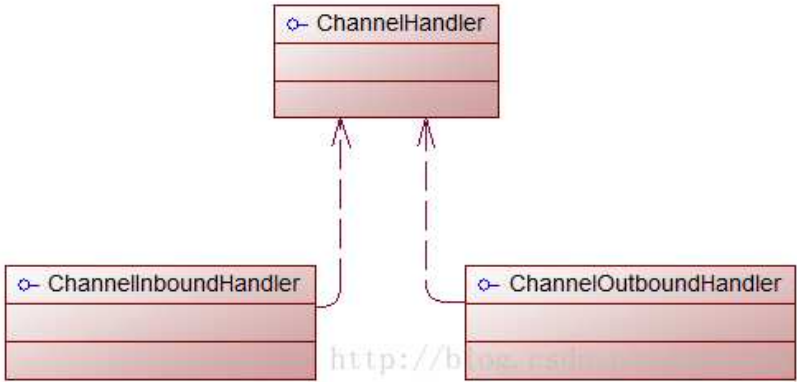
Netty允许处理IO和接受连接使用同一个EventLoopGroup，这在实际中适用于多种应用。上图显示了一个EventLoopGroup处理连接请求和IO操作。

下一节我们将介绍Netty是如何执行IO操作以及在什么时候执行。

3.4 Channel Handlers and Data Flow(通道处理和数据流)

本节我们一起来看看当你发送或接收数据时发生了什么？回想本章开始提到的handler概念。要明白Netty程序write或read时发生了什么，首先要对Handler是什么有一定的了解。Handlers自身依赖于ChannelPipeline来决定它们执行的顺序，因此不可能通过ChannelPipeline定义处理程序的某些方面,反过来不可能定义也不可能通过ChannelHandler定义ChannelPipeline的某些方面。没必要说我们必须定义一个自己和其他的规定。本节将介绍ChannelHandler和ChannelPipeline在某种程度上细微的依赖。

在很多地方，Netty的ChannelHandler是你的应用程序中处理最多的。即使你没有意识到这一点，若果你使用Netty应用将至少有一个ChannelHandler参与，换句话说，ChannelHandler对很多事情是关键。那么ChannelHandler究竟是什么？给ChannelHandler一个定义不容易，我们可以理解为ChannelHandler是一段执行业务逻辑处理数据的代码，它们来来往往的通过ChannelPipeline。实际上，ChannelHandler是定义一个handler的父接口，ChannelInboundHandler和ChannelOutboundHandler都实现ChannelHandler接口，如下图：

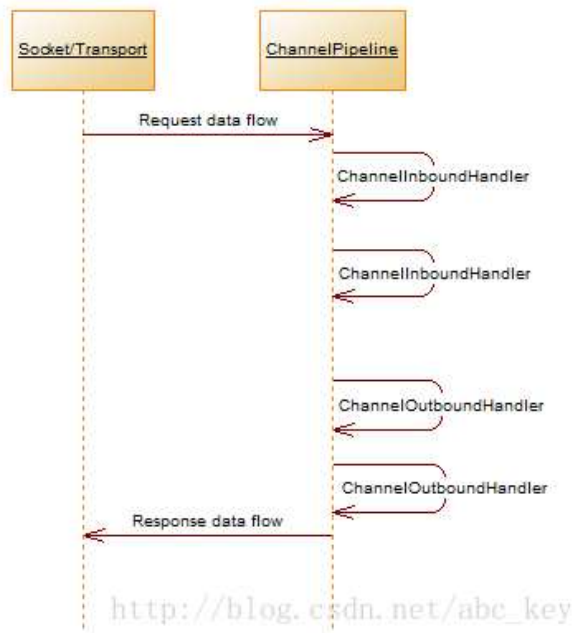


上图显示的比较容易，更重要的是ChannelHandler在数据流方面的应用，在这里讨论的例子只是一个简单的例子。ChannelHandler被应用在许多方面，在本书中会慢慢学习。

Netty中有两个方向的数据流，上图显示的入站(ChannelInboundHandler)和出站(ChannelOutboundHandler)之间有一个明显的区别：若数据是从用户应用程序到远程主机则是“出站(outbound)”，相反若数据时从远程主机到用户应用程序则是“入站(inbound)”。

为了使数据从一端到达另一端，一个或多个ChannelHandler将以某种方式操作数据。这些ChannelHandler会在程序的“引导”阶段被添加ChannelPipeline中，并且被添加的顺序将决定处理数据的顺序。ChannelPipeline的作用我们可以理解为用来管理ChannelHandler的一个容器，每个ChannelHandler处理各自的数据(例如入站数据只能由ChannelInboundHandler处理)，处理完成后将转换的数据放到ChannelPipeline中交给下一个ChannelHandler继续处理，直到最后一个ChannelHandler处理完成。

下图显示了ChannelPipeline的处理过程：



上图显示ChannelInboundHandler和ChannelOutboundHandler都要经过相同的ChannelPipeline。

在ChannelPipeline中，如果消息被读取或有任何其他的入站事件，消息将从ChannelPipeline的头部开始传递给第一个ChannelInboundHandler，这个ChannelInboundHandler可以处理该消息或将消息传递到下一个ChannelInboundHandler中，一旦在ChannelPipeline中没有剩余的ChannelInboundHandler后，ChannelPipeline就知道消息已被所有的饿Handler处理完成了。

反过来也是如此，任何出站事件或写入将从ChannelPipeline的尾部开始，并传递到最后一个ChannelOutboundHandler。ChannelOutboundHandler的作用和ChannelInboundHandler相同，它可以传递事件消息到下一个Handler或者自己处理消息。不同的是ChannelOutboundHandler是从ChannelPipeline的尾部开始，而ChannelInboundHandler是从ChannelPipeline的头部开始，当处理完第一个ChannelOutboundHandler处理完成后会出发一些操作，比如一个写操作。

一个事件能传递到下一个ChannelInboundHandler或上一个ChannelOutboundHandler，在ChannelPipeline中通过使用ChannelHandlerContext调用每一个方法。Netty提供了抽象的事件基类称为ChannelInboundHandlerAdapter和ChannelOutboundHandlerAdapter。每个都提供了在ChannelPipeline中通过调用相应的方法将事件传递给下一个Handler的方法的实现。我们能覆盖的方法就是我们需要的处理。

可能有读者会奇怪，出站和入站的操作不同，能放在同一个ChannelPipeline工作？Netty的设计是很巧妙的，入站和出站Handler有不同的实现，Netty能跳过一个不能处理的操作，所以在出站事件的情况下，ChannelInboundHandler将被跳过，Netty知道每个handler都必须实现ChannelInboundHandler或ChannelOutboundHandler。

当一个ChannelHandler添加到ChannelPipeline中时获得一个ChannelHandlerContext。通常是安全的获得这个对象的引用，但是当一个数据报协议如UDP时这是不正确的，这个对象可以在之后用来获取底层通道，因为要用它来read/write消息，因此通道会保留。也就是说Netty中发送消息有两种方法：直接写入通道或写入ChannelHandlerContext对象。这两种方法的主要区别如下：

- 直接写入通道导致处理消息从ChannelPipeline的尾部开始
- 写入ChannelHandlerContext对象导致处理消息从ChannelPipeline的下一个handler开始

3.5 编码器、解码器和业务逻辑：细看Handlers

如前面所说，有很多不同类型的handlers，每个handler的依赖于它们的基类。Netty提供了一系列的“Adapter”类，这让事情变的很简单。每个handler负责转发时间到ChannelPipeline的下一个handler。在*Adapter类(和子类)中是自动完成的，因此我们只需要在感兴趣的*Adpater中重写方法。这些功能可以帮助我们非常简单的编码/解码消息。有几个适配器(adapter)允许自定义ChannelHandler，一般自定义ChannelHandler需要继承编码/解码适配器类中的一个。Netty有一下适配器：

- ChannelHandlerAdapter

- ChannelInboundHandlerAdapter
- ChannelOutboundHandlerAdapter

三个ChannelHandler类，我们重点看看encoder,decoder和SimpleChannelInboundHandler<I>，SimpleChannelInboundHandler<I>继承ChannelInboundHandlerAdapter。

3.5.1 Encoders(编码器), decoders(解码器)

发送或接收消息后，Netty必须将消息数据从一种形式转化为另一种。接收消息后，需要将消息从字节码转成Java对象(由某种解码器解码)；发送消息前，需要将Java对象转成字节(由某些类型的编码器进行编码)。这种转换一般发生在网络程序中，因为网络上只能传输字节数据。

有多种基础类型的编码器和解码器，要使用哪种取决于想实现的功能。要弄清楚某种类型的编解码器，从类名就可以看出，如“ByteToMessageDecoder”、“MessageToByteEncoder”，还有Google的协议“ProtobufEncoder”和“ProtobufDecoder”。

严格的说其他handlers可以做编码器和适配器，使用不同的Adapter classes取决你想要做什么。如果是解码器则有一个ChannelInboundHandlerAdapter或ChannelInboundHandler，所有的解码器都继承或实现它们。“channelRead”方法/事件被覆盖，这个方法从入站(inbound)通道读取每个消息。重写的channelRead方法将调用每个解码器的“decode”方法并通过ChannelHandlerContext.fireChannelRead(Object msg)传递给ChannelPipeline中的下一个ChannelInboundHandler。

类似入站消息，当你发送一个消息出去(出站)时，除编码器将消息转成字节码外还会转发到下一个ChannelOutboundHandler。

3.5.2 业务逻辑(Domain logic)

也许最常见的是应用程序处理接收到消息后进行解码，然后供相关业务逻辑模块使用。所以应用程序只需要扩展SimpleChannelInboundHandler<I>，也就是我们自定义一个继承SimpleChannelInboundHandler<I>的handler类，其中<I>是handler可以处理的消息类型。通过重写父类的方法可以获得一个ChannelHandlerContext的引用，它们接受一个ChannelHandlerContext的参数，你可以在class中当一个属性存储。

处理程序关注的主要方法是“channelRead0(ChannelHandlerContext ctx, I msg)”，每当Netty调用这个方法，对象“I”是消息，这里使用了Java的泛型设计，程序就能处理I。如何处理消息完全取决于程序的需要。在处理消息时有一点需要注意的，在Netty中事件处理IO一般有很多线程，程序中尽量不要阻塞IO线程，因为阻塞会降低程序的性能。

必须不阻塞IO线程意味着在ChannelHandler中使用阻塞操作会有问题。幸运的是Netty提供了解决方案，我们可以在添加ChannelHandler到ChannelPipeline中时指定一个EventExecutorGroup，EventExecutorGroup会获得一个EventExecutor，EventExecutor将执行ChannelHandler的所有方法。EventExecutor将使用不同的线程来执行和释放EventLoop。

Netty In Action中文版 - 第四章：Transports(传输)

- 目录(?)

- 1. 案例研究切换传输方式
 - 1. 使用Java的IO和NIO
 - 2. Netty中使用IO和NIO
 - 3. Netty中实现异步支持
- 2. Transport API
- 3. Netty包含的传输实现
 - 1. NIO - Nonblocking IO
 - 2. OIO - Old blocking IO
 - 3. Local - In VM transport
 - 4. Embedded transport
- 4. 每种传输方式在什么时候使用

本章内容

- Transports(传输)
- NIO(non-blocking IO,New IO), OIO(Old IO,blocking IO), Local(本地), Embedded(嵌入式)
- Use-case(用例)
- APIs(接口)

网络应用程序一个很重要的工作是传输数据。传输数据的过程不一样取决是使用哪种交通工具，但是传输的方式是一样的：都是以字节码传输。**Java**开发网络程序传输数据的过程和方式是被抽象了的，我们不需要关注底层接口，只需要使用**Java API**或其他网络框架如**Netty**就能达到传输数据的目的。发送数据和接收数据都是字节码。**Nothing more,nothing less.**

如果你曾经使用Java提供的网络接口工作过，你可能已经遇到过想从阻塞传输切换到非阻塞传输的情况，这种切换是比较困难的，因为阻塞IO和非阻塞IO使用的API有很大的差异；Netty提供了上层的传输实现接口使得这种情况变得简单。我们可以让所写的代码尽可能通用，而不会依赖一些实现相关的APIs。当我们想切换传输方式的时候不需要花很大的精力和时间来重构代码。

本章将介绍统一的API以及如何使用它们，会拿Netty的API和Java的API做比较来告诉你为什么Netty可以更容易的使用。本章也提供了一些优质的用例代码，以便最佳使用Netty。使用Netty不需要其他的网络框架或网络编程经验，若有则只是对理解netty有帮助，但不是必要的。下面让我们来看看真是世界里的传输工作。

4.1 案例研究：切换传输方式

为了让你想象如何运输，我会从一个简单的应用程序开始，这个应用程序什么都不做，只是接受客户端连接并发送“Hi!”字符串消息到客户端，发送完了就断开连接。我不会详细讲解这个过程的实现，它只是一个例子。

4.1.1 使用Java的I/O和NIO

我们将不用Netty实现这个例子，下面代码是使用阻塞IO实现的例子：

```
[java]  
```

```
01. package netty.in.action;
02.
03. import java.io.IOException;
04. import java.io.OutputStream;
05. import java.net.ServerSocket;
06. import java.net.Socket;
07. import java.nio.charset.Charset;
08.
09. /**
10.  * Blocking networking without Netty
11.  * @author c.k
12.  *
13.  */
14. public class PlainOoServer {
15.
16.     public void server(int port) throws Exception {
17.         //bind server to port
18.         final ServerSocket socket = new ServerSocket(port);
19.         try {
20.             while(true){
21.                 //accept connection
22.                 final Socket clientSocket = socket.accept();
23.                 System.out.println("Accepted connection from " + clientSocket);
24.                 //create new thread to handle connection
25.                 new Thread(new Runnable() {
26.                     @Override
```



```
27.         public void run() {
28.             Out put St ream out ;
29.             try{
30.                 out = clientSocket . get Out put St ream();
31.                 //write message to connected client
32.                 out . write( "H !\r\n". get Bytes( Char set . for Name( "UTF- 8" )));
33.                 out . flush();
34.                 //close connection once message written and flushed
35.                 clientSocket . close();
36.             } catch (IOException e) {
37.                 try {
38.                     clientSocket . close();
39.                 } catch (IOException e1) {
40.                     e1 . print St ackTrace();
41.                 }
42.             }
43.         }
44.     }). start(); //start thread to begin handling
45. }
46. } catch (Exception e) {
47.     e . print St ackTrace();
48.     socket . close();
49. }
50. }
51.
52. }
```

上面的方式很简洁，但是这种阻塞模式在大连接数的情况就会有严重的问题，如客户端连接超时，服务器响应严重延迟。为了解决这种情况，我们可以使用异步网络处理所有的并发连接，但问题在于NIO和OIO的API是完全不同的，所以一个用OIO开发的网络应用程序想要使用NIO重构代码几乎是重新开发。

下面代码是使用Java NIO实现的例子：

```
[ java ]
01. package netty.in.action;
02.
03. import java.net.InetSocketAddress;
04. import java.net.ServerSocket;
05. import java.nio.ByteBuffer;
06. import java.nio.channels.SelectionKey;
07. import java.nio.channels.Selector;
08. import java.nio.channels.ServerSocketChannel;
09. import java.nio.channels.SocketChannel;
10. import java.util.Iterator;
11. /**
12.  * Asynchronous networking without Netty
13.  * @author c.k
14.  *
15.  */
16. public class PlainNoServer {
17.
18.     public void server(int port) throws Exception {
19.         System.out.println("Listening for connections on port " + port);
20.         //open Selector that handles channels
21.         Selector selector = Selector.open();
22.         //open ServerSocketChannel
23.         ServerSocketChannel serverChannel = ServerSocketChannel.open();
24.         //get ServerSocket
25.         ServerSocket serverSocket = serverChannel.socket();
26.         //bind server to port
27.         serverSocket.bind(new InetSocketAddress(port));
28.         //set to non-blocking
29.         serverChannel.configureBlocking(false);
30.         //register ServerSocket to selector and specify that it is interested in new accepted clients
31.         serverChannel.register(selector, SelectionKey.OP_ACCEPT);
32.         final ByteBuffer msg = ByteBuffer.wrap("Hi!\r\n".getBytes());
33.         while (true) {
34.             //Wait for new events that are ready for process. This will block until something happens
35.             int n = selector.select();
36.             if (n > 0) {
37.                 //Obtain all SelectionKey instances that received events
38.                 Iterator<SelectionKey> iter = selector.selectedKeys().iterator();
39.                 while (iter.hasNext()) {
40.                     SelectionKey key = iter.next();
41.                     iter.remove();
42.                     try {
43.                         //Check if event was because new client ready to get accepted
```

```
44.         if (key.isAcceptable()) {
45.             ServerSocketChannel server = (ServerSocketChannel) key.channel();
46.             SocketChannel client = server.accept();
47.             System.out.println("Accepted connection from" + client);
48.             client.configureBlocking(false);
49.             //Accept client and register it to selector
50.             client.register(selector, SelectionKey.OP_WRITE, msg.duplicate());
51.         }
52.         //Check if event was because socket is ready to write data
53.         if (key.isWritable()) {
54.             SocketChannel client = (SocketChannel) key.channel();
55.             ByteBuffer buff = (ByteBuffer) key.attachment();
56.             //write data to connected client
57.             while (buff.hasRemaining()) {
58.                 if (client.write(buff) == 0) {
59.                     break;
60.                 }
61.             }
62.             client.close();//close client
63.         }
64.     } catch (Exception e) {
65.         key.cancel();
66.         key.channel().close();
67.     }
68. }
69. }
70. }
71. }
72.
73. }
```

如你所见，即使它们实现的功能是一样，但是代码完全不同。下面我们将用Netty来实现相同的功能。

4.1.2 Netty中使用I/O和NIO

下面代码是使用Netty作为网络框架编写的一个阻塞IO例子：

```
[ java ]
01. package netty.in.action;
02.
03. import java.net.InetSocketAddress;
04.
05. import io.netty.bootstrap.ServerBootstrap;
06. import io.netty.buffer.ByteBuf;
07. import io.netty.buffer.Unpooled;
08. import io.netty.channel.Channel;
09. import io.netty.channel.ChannelFuture;
10. import io.netty.channel.ChannelFutureListener;
11. import io.netty.channel.ChannelHandlerContext;
12. import io.netty.channel.ChannelInboundHandlerAdapter;
13. import io.netty.channel.ChannelInitializer;
14. import io.netty.channel.EventLoopGroup;
15. import io.netty.channel.nio.NioEventLoopGroup;
16. import io.netty.channel.socket.nio.NioServerSocketChannel;
17. import io.netty.util.CharsetUtil;
18.
19. public class NettyNioServer {
20.
21.     public void server(int port) throws Exception {
22.         final ByteBuf buf = Unpooled.unreleasableBuffer(Unpooled.copiedBuffer("Hi!\r\n", CharsetUtil.UTF_8));
23.         // 事件循环组
24.         EventLoopGroup group = new NioEventLoopGroup();
25.         try {
26.             // 用来引导服务器配置
27.             ServerBootstrap b = new ServerBootstrap();
28.             // 使用NIO阻塞模式
29.             b.group(group).channel(NioServerSocketChannel.class).localAddress(new InetSocketAddress(port))
30.             // 指定ChannelInitializer初始化handlers
31.             .childHandler(new ChannelInitializer<Channel>() {
32.                 @Override
33.                 protected void initChannel(Channel ch) throws Exception {
34.                     // 添加一个“入站”handler到ChannelPipeline
35.                     ch.pipeline().addLast(new ChannelInboundHandlerAdapter() {
36.                         @Override
37.                         public void channelActive(ChannelHandlerContext ctx) throws Exception {
38.                             // 连接后，写消息到客户端，写完后便关闭连接
39.                             ctx.writeAndFlush(buf.duplicate()).addListener(ChannelFutureListener.CLOSE);
40.                         }
41.                     });
42.                 }
43.             });
44.         } catch (Exception e) {
45.             group.shutdownGracefully();
46.         }
47.     }
48. }
```

```
42.         }
43.     });
44.     // 绑定服务器接受连接
45.     ChannelFuture f = b.bind().sync();
46.     f.channel().closeFuture().sync();
47. } catch (Exception e) {
48.     // 释放所有资源
49.     group.shutdownGracefully();
50. }
51. }
52.
53. }
```

上面代码实现功能一样，但结构清晰明了，这只是Netty的优势之一。

4.1.3 Netty中实现异步支持

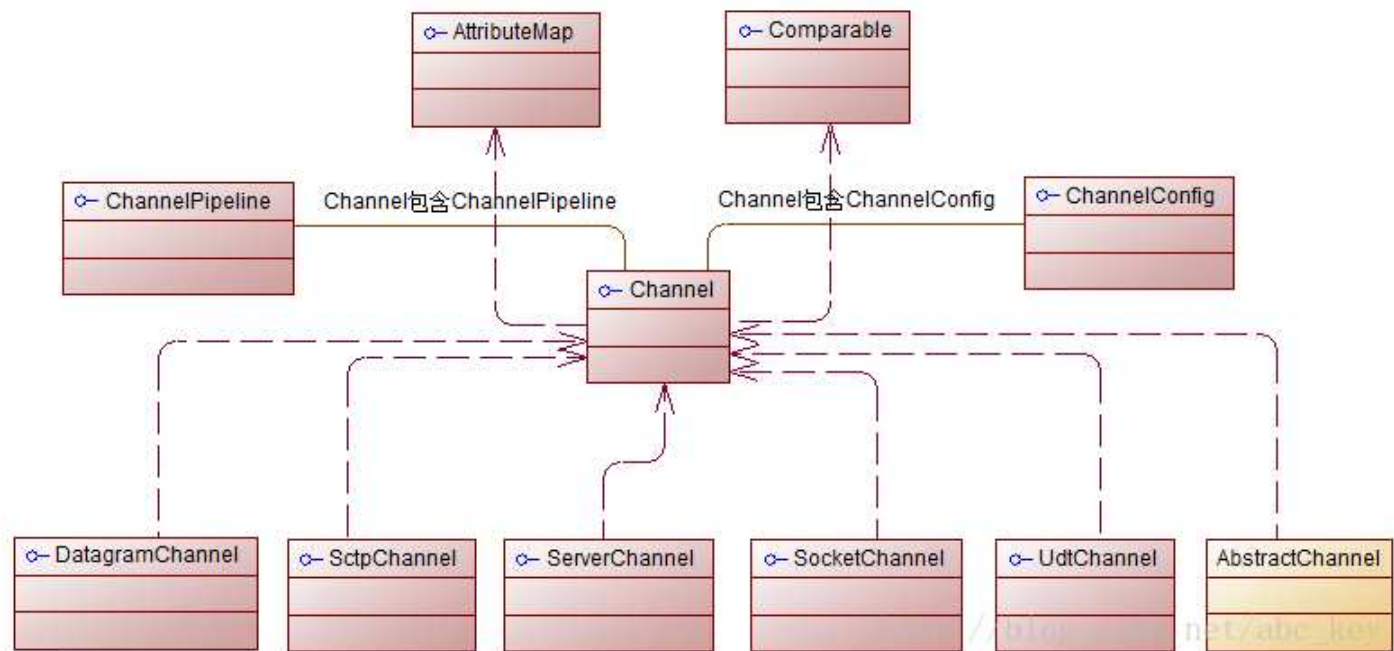
下面代码是使用Netty实现异步，可以看出使用Netty由OIO切换到NIO是非常的方便。

```
[java]
01. package netty.in.action;
02.
03. import io.netty.bootstrap.ServerBootstrap;
04. import io.netty.buffer.ByteBuf;
05. import io.netty.buffer.Unpooled;
06. import io.netty.channel.ChannelFuture;
07. import io.netty.channel.ChannelFutureListener;
08. import io.netty.channel.ChannelHandlerContext;
09. import io.netty.channel.ChannelInboundHandlerAdapter;
10. import io.netty.channel.ChannelInitializer;
11. import io.netty.channel.EventLoopGroup;
12. import io.netty.channel.nio.NioEventLoopGroup;
13. import io.netty.channel.socket.SocketChannel;
14. import io.netty.channel.socket.nio.NioServerSocketChannel;
15. import io.netty.util.CharsetUtil;
16.
17. import java.net.InetSocketAddress;
18.
19. public class NettyNioServer {
20.
21.     public void server(int port) throws Exception {
22.         final ByteBuf buf = Unpooled.unreleasableBuffer(Unpooled.copiedBuffer("H!\r\n", CharsetUtil.UTF_8));
23.         // 事件循环组
24.         EventLoopGroup group = new NioEventLoopGroup();
25.         try {
26.             // 用来引导服务器配置
27.             ServerBootstrap b = new ServerBootstrap();
28.             // 使用NIO异步模式
29.             b.group(group).channel(NioServerSocketChannel.class).localAddress(new InetSocketAddress(port))
30.             // 指定ChannelInitializer初始化handlers
31.             .childHandler(new ChannelInitializer<SocketChannel>() {
32.                 @Override
33.                 protected void initChannel(SocketChannel ch) throws Exception {
34.                     // 添加一个“入站”handler到ChannelPipeline
35.                     ch.pipeline().addLast(new ChannelInboundHandlerAdapter() {
36.                         @Override
37.                         public void channelActive(ChannelHandlerContext ctx) throws Exception {
38.                             // 连接后，写消息到客户端，写完后便关闭连接
39.                             ctx.writeAndFlush(buf.duplicate()).addListener(ChannelFutureListener.CLOSE);
40.                         }
41.                     });
42.                 }
43.             });
44.             // 绑定服务器接受连接
45.             ChannelFuture f = b.bind().sync();
46.             f.channel().closeFuture().sync();
47.         } catch (Exception e) {
48.             // 释放所有资源
49.             group.shutdownGracefully();
50.         }
51.     }
52. }
```

因为Netty使用相同的API来实现每个传输，它并不关心你使用什么来实现。Netty通过操作Channel接口和ChannelPipeline、ChannelHandler来实现传输。

4.2 Transport API

传输API的核心是Channel接口，它用于所有出站的操作。Channel接口的类层次结构如下



如上图所示，每个Channel都会分配一个ChannelPipeline和ChannelConfig。ChannelConfig负责设置并存储配置，并允许在运行期间更新它们。传输一般有特定的配置设置，只作用于传输，没有其他的实现。ChannelPipeline容纳了使用的ChannelHandler实例，这些ChannelHandler将处理通道传递的“入站”和“出站”数据。ChannelHandler的实现允许你改变数据状态和传输数据，本书有章节详细讲解ChannelHandler，ChannelHandler是Netty的重点概念。

现在我们可以使用ChannelHandler做下面一些事情：

- 传输数据时，将数据从一种格式转换到另一种格式
- 异常通知
- Channel变为有效或无效时获得通知
- Channel被注册或从EventLoop中注销时获得通知
- 通知用户特定事件

这些ChannelHandler实例添加到ChannelPipeline中，在ChannelPipeline中按顺序逐个执行。它类似于一个链条，有使用过Servlet的读者可能会更容易理解。

ChannelPipeline实现了拦截过滤器模式，这意味着我们连接不同的ChannelHandler来拦截并处理经过ChannelPipeline的数据或事件。可以把ChannelPipeline想象成UNIX管道，它允许不同的命令链(ChannelHandler相当于命令)。你还可以在运行时根据需要添加ChannelHandler实例到ChannelPipeline或从ChannelPipeline中删除，这能帮助我们构建高度灵活的Netty程序。此外，访问指定的ChannelPipeline和ChannelConfig，你能在Channel自身上进行操作。Channel提供了很多方法，如下列表：

- eventLoop()，返回分配给Channel的EventLoop
- pipeline()，返回分配给Channel的ChannelPipeline
- isActive()，返回Channel是否激活，已激活说明与远程连接对等
- localAddress()，返回已绑定的本地SocketAddress
- remoteAddress()，返回已绑定的远程SocketAddress
- write()，写数据到远程客户端，数据通过ChannelPipeline传输过去

后面会越来越熟悉这些方法，现在只需要记住我们的操作都是在相同的接口上运行，Netty的高灵活性让你可以以不同的传输实现进行重构。

写数据到远程已连接客户端可以调用Channel.write()方法，如下代码：

```
[java]
01. Channel channel = ...
02. // Create ByteBuffer that holds data to write
03. ByteBuffer buf = Unpooled.copiedBuffer("your data", CharsetUtil.UTF_8);
04. // Write data
05. ChannelFuture cf = channel.write(buf);
06. // Add ChannelFutureListener to get notified after write completes
07. cf.addListener(new ChannelFutureListener() {
08.     @Override
09.     public void operationComplete(ChannelFuture future) {
10.         // Write operation completes without error
11.         if (future.isSuccess()) {
12.             System.out.println("Write successful.");
13.         } else {
14.             // Write operation completed but because of error
15.             System.err.println("Write error.");
16.             future.cause().printStackTrace();
17.         }
18.     }
19. });
```

```
17.         }
18.     }
19. });
```

Channel是线程安全(thread-safe)的，它可以被多个不同的线程安全的操作，在多线程环境下，所有的方法都是安全的。正因为Channel是安全的，我们存储对Channel的引用，并在学习的时候使用它写入数据到远程已连接的客户端，使用多线程也是如此。下面的代码是一个简单的多线程例子：

```
[java]

01. final Channel channel = ...
02. // Create ByteBuffer that holds data to write
03. final ByteBuffer buf = Unpooled.copiedBuffer("your data", CharsetUtil.UTF_8);
04. // Create Runnable which writes data to channel
05. Runnable writer = new Runnable() {
06.     @Override
07.     public void run() {
08.         channel.write(buf.duplicate());
09.     }
10. };
11. // Obtain reference to the Executor which uses threads to execute tasks
12. Executor executor = Executors.newCachedThreadPool();
13. // write in one thread
14. // Hand over write task to executor for execution in thread
15. executor.execute(writer);
16. // write in another thread
17. // Hand over another write task to executor for execution in thread
18. executor.execute(writer);
```

此外，这种方法保证了写入的消息以相同的顺序通过写入它们的方法。想了解所有方法的使用可以参考Netty API文档。

4.3 Netty包含的传输实现

Netty自带了一些传输协议的实现，虽然没有支持所有的传输协议，但是其自带的已足够我们来使用。Netty应用程序的传输协议依赖于底层协议，本节我们将学习Netty中的传输协议。

Netty中的传输方式有如下几种：

- NIO, io.netty.channel.socket.nio，基于java.nio.channels的工具包，使用选择器作为基础的方法。
- OIO, io.netty.channel.socket.oio，基于java.net的工具包，使用阻塞流。
- Local, io.netty.channel.local，用来在虚拟机之间本地通信。
- Embedded, io.netty.channel.embedded，嵌入传输，它允许在没有真正网络的运输中使用ChannelHandler，可以非常有用的来测试ChannelHandler的实现。

4.3.1 NIO - Nonblocking I/O

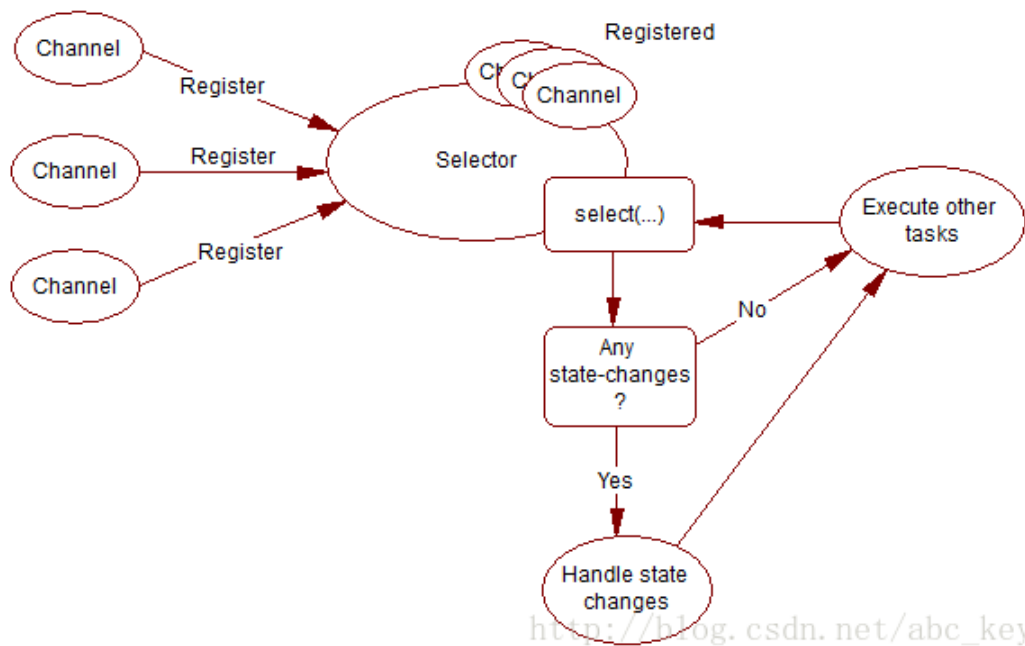
NIO传输是目前最常用的方式，它通过使用选择器提供了完全异步的方式操作所有的I/O，NIO从Java 1.4才被提供。NIO中，我们可以注册一个通道或获得某个通道的改变的状态，通道状态有下面几种改变：

- 一个新的Channel被接受并已准备好
- Channel连接完成
- Channel中有数据并已准备好读取
- Channel发送数据出去

处理完改变的状态后需重新设置他们的状态，用一个线程来检查是否有已准备好的Channel，如果有则执行相关事件。在这里可能只同时一个注册的事件而忽略其他的。选择器所支持的操作在SelectionKey中定义，具体如下：

- OP_ACCEPT，有新连接时得到通知
- OP_CONNECT，连接完成后得到通知
- OP_READ，准备好读取数据时得到通知
- OP_WRITE，写入数据到通道时得到通知

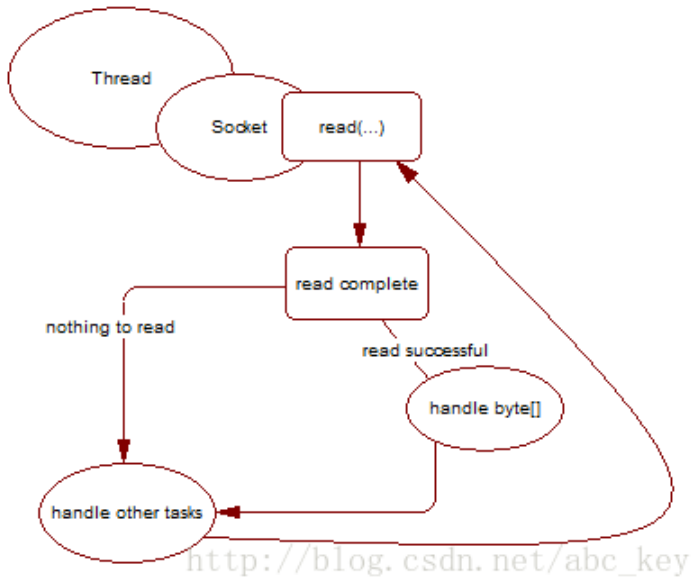
Netty中的NIO传输就是基于这样的模型来接收和发送数据，通过封装将自己的接口提供给用户使用，这完全隐藏了内部实现。如前面所说，Netty隐藏内部的实现细节，将抽象出来的API暴露出来供使用，下面是处理流程图：



NIO在处理过程也会有一定的延迟，若连接数不大的话，延迟一般在毫秒级，但是其吞吐量依然比OIO模式的要高。Netty中的NIO传输是“zero-file-copy”，也就是零文件复制，这种机制可以让程序速度更快，更高效的从文件系统中传输内容，零复制就是我们的应用程序不会将发送的数据先复制到JVM堆栈在进行处理，而是直接从内核空间操作。接下来我们将讨论OIO传输，它是阻塞的。

4.3.2 OIO - Old blocking I/O

OIO就是java中提供的Socket接口，java最开始只提供了阻塞的Socket，阻塞会导致程序性能低。下面是OIO的处理流程图，若想详细了解，可以参阅其他相关资料。



4.3.3 Local - In VM transport

Netty包含了本地传输，这个传输实现使用相同的API用于虚拟机之间的通信，传输是完全异步的。每个Channel使用唯一的SocketAddress，客户端通过使用SocketAddress进行连接，在服务器会被注册为长期运行，一旦通道关闭，它会自动注销，客户端无法再使用它。

连接到本地传输服务器的行为与其他的传输实现几乎是相同的，需要注意的一个重点是只能在本地的服务器和客户端上使用它们。Local未绑定任何Socket，值提供JVM进程之间的通信。

4.3.4 Embedded transport

Netty还包括嵌入传输，与之前讲述的其他传输实现比较，它是不是一个真的传输呢？若不是一个真的传输，我们用它可以做什么呢？Embedded transport允许更容易的使用不同的ChannelHandler之间的交互，这也更容易嵌入到其他的ChannelHandler实例并像一个辅助类一样使用它们。它一般用来测试特定的ChannelHandler实现，也可以在ChannelHandler中重新使用一些ChannelHandler来进行扩展，为了实现这样的目的，它自带了一个具体的Channel实现，即：EmbeddedChannel。

4.4 每种传输方式在什么时候使用？

不多加赘述，看下面列表：

- OIO，在低连接数、需要低延迟时、阻塞时使用
- NIO，在高连接数时使用
- Local，在同一个JVM内通信时使用
- Embedded，测试ChannelHandler时使用

目录(?)

[]

1. Buffer API

2. ByteBuf - 字节数据容器

1. ByteBuf如何在工作

2. 不同类型的ByteBuf

3. ByteBuf的字节操作

1. 随机访问索引

2. 顺序访问索引

3. Discardable bytes废弃字节

4. 可读字节实际内容

5. 可写字节Writable bytes

6. 清除缓冲区索引Clearing the buffer indexes

7. 搜索操作Search operations

8. 标准和重置Mark and reset

9. 衍生的缓冲区Derived buffers

10. 读写操作以及其他一些操作

4. ByteBufHolder

1. ByteBufAllocator

2. Unpooled

3. ByteBufUtil

5. Summary

本章介绍

- ByteBuf
- ByteBufHolder
- ByteBufAllocator
- 使用这些接口分配缓冲和执行操作

每当你需要传输数据时，它必须包含一个缓冲区。Java NIO API自带的缓冲区类是相当有限的，没有经过优化，使用JDK的ByteBuffer操作更复杂。缓冲区是一个重要的组建，它是API的一部分。Netty提供了一个强大的缓冲区实现用于表示一个字节序列，并帮助你操作原始字节或自定义的POJO。Netty的ByteBuf相当于JDK的ByteBuffer，ByteBuf的作用是在Netty中通过Channel传输数据。它被重新设计以解决JDK的ByteBuffer中的一些问题，从而使开发人员开发网络应用程序显得更有效率。本章将讲述Netty中的缓冲区，并了解它为什么比JDK自带的缓冲区实现更优秀，还会深入了解在Netty中使用ByteBuf访问数据以及如何使用它。

5.1 Buffer API

Netty的缓冲API有两个接口：

- ByteBuf
- ByteBufHolder

Netty使用reference-counting(引用计数)的时候知道安全释放Buf和其他资源，虽然知道Netty有效的使用引用计数，这都是自动完成的。这允许Netty使用池和其他技巧来加快速度和保持内存利用率在正常水平，你不需要做任何事情来实现这一点，但是在开发Netty应用程序时，你应该处理数据尽快释放池资源。

Netty缓冲API提供了几个优势：

- 可以自定义缓冲类型
- 通过一个内置的复合缓冲类型实现零拷贝
- 扩展性好，比如StringBuffer
- 不需要调用flip()来切换读/写模式
- 读取和写入索引分开
- 方法链
- 引用计数
- Pooling(池)

5.2 ByteBuf - 字节数据容器

当需要与远程进行交互时，需要以字节码发送/接收数据。由于各种原因，一个高效、方便、易用的数据接口是必须的，而Netty的ByteBuf满足这些需求，ByteBuf是一个很好的经过优化的数据容器，我们可以将字节数据有效的添加到ByteBuf中或从ByteBuf中获取数据。ByteBuf有2部分：一个用于读，一个用于写。我们可以按顺序的读取数据，并且可以跳到开始重新读一遍。所有的数据操作，我们只需要做的是调整读取数据索引和再次开始读操作。

5.2.1 ByteBuf如何在工作？

写入数据到ByteBuf后，写入索引是增加的字节数量。开始读字节后，读取索引增加。你可以读取字节，直到写入索引和读取索引处理相同的位置，次数若继续读取，则会抛出IndexOutOfBoundsException。调用ByteBuf的任何方法开始读/写都会单独维护读索引和写索引。ByteBuf的默认最大容量限制是Integer.MAX_VALUE，写入时若超出这个值将会导致一个异常。

ByteBuf类似于一个字节数组，最大的区别是读和写的索引可以用来控制对缓冲区数据的访问。下图显示了一个容量为16的ByteBuf：



5.2.2 不同类型的ByteBuf

使用Netty时会遇到3种不同类型的ByteBuf

Heap Buffer(堆缓冲区)

最常用的类型是ByteBuf将数据存储在JVM的堆空间，这是通过将数据存储在数组的实现。堆缓冲区可以快速分配，当不使用时也可以快速释放。它还提供了直接访问数组的方法，通过ByteBuf.array()来获取byte[]数据。

访问非堆缓冲区ByteBuf的数组会导致UnsupportedOperationException，可以使用ByteBuf.hasArray()来检查是否支持访问数组。

Direct Buffer(直接缓冲区)

直接缓冲区，在堆之外直接分配内存。直接缓冲区不会占用堆空间容量，使用时应该考虑到应用程序要使用的最大内存容量以及如何限制它。直接缓冲区在使用Socket传递数据时性能很好，因为若使用间接缓冲区，JVM会先将数据复制到直接缓冲区再进行传递；但是直接缓冲区的缺点是在分配内存空间和释放内存时比堆缓冲区更复杂，而Netty使用内存池来解决这样的问题，这也是Netty使用内存池的原因之一。直接缓冲区不支持数组访问数据，但是我们可以间接的访问数据数组，如下面代码：

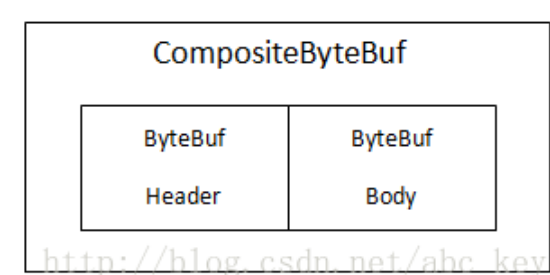
```
[ java ]
01. ByteBuf directBuf = Unpooled.directBuffer(16);
02. if(!directBuf.hasArray()){
03.     int len = directBuf.readableBytes();
04.     byte[] arr = new byte[len];
05.     directBuf.getBytes(0, arr);
06. }
```

访问直接缓冲区的数据数组需要更多的编码和更复杂的操作，建议若需要在数组访问数据使用堆缓冲区会更好。

Composite Buffer(复合缓冲区)

复合缓冲区，我们可以创建多个不同的ByteBuf，然后提供一个这些ByteBuf组合的视图。复合缓冲区就像一个列表，我们可以动态的添加和删除其中的ByteBuf，JDK的ByteBuffer没有这样的功能。Netty提供了CompositeByteBuf类来处理复合缓冲区，CompositeByteBuf只是一个视图，CompositeByteBuf.hasArray()总是返回false，因为它可能包含一些直接或间接的不同类型的ByteBuf。

例如，一条消息由header和body两部分组成，将header和body组装成一条消息发送出去，可能body相同，只是header不同，使用CompositeByteBuf就不用每次都重新分配一个新的缓冲区。下图显示CompositeByteBuf组成header和body：



若使用JDK的ByteBuffer就不能这样简单的实现，只能创建一个数组或创建一个新的ByteBuffer，再将内容复制到新的ByteBuffer中。下面是使用CompositeByteBuf的例子：

```
[ java ]
01. CompositeByteBuf compBuf = Unpooled.compositeBuffer();
02. ByteBuf heapBuf = Unpooled.buffer(8);
03. ByteBuf directBuf = Unpooled.directBuffer(16);
04. // 添加ByteBuf到CompositeByteBuf
05. compBuf.addComponent(heapBuf, directBuf);
06. // 删除第一个ByteBuf
07. compBuf.removeComponent(0);
```

```
08. Iterator<ByteBuf> iter = compBuf.iterator();
09. while(iter.hasNext()){
10.     System.out.println(iter.next().toString());
11. }
12. // 使用数组访问数据
13. if(!compBuf.hasArray()){
14.     int len = compBuf.readableBytes();
15.     byte[] arr = new byte[len];
16.     compBuf.getBytes(0, arr);
17. }
```

CompositeByteBuf是ByteBuf的子类，我们可以像操作BytBuf一样操作CompositeByteBuf。并且Netty优化套接字读写的操作是尽可能的使用CompositeByteBuf来做的，使用CompositeByteBuf不会操作内存泄露问题。

5.3 ByteBuf的字节操作

ByteBuf提供了许多操作，允许修改其中的数据内容或只是读取数据。ByteBuf和JDK的ByteBuffer很像，但是ByteBuf提供了更好的性能。

5.3.1 随机访问索引

ByteBuf使用zero-based-indexing(从0开始的索引)，第一个字节的索引是0，最后一个字节的索引是ByteBuf的capacity - 1，下面代码是遍历ByteBuf的所有字节：

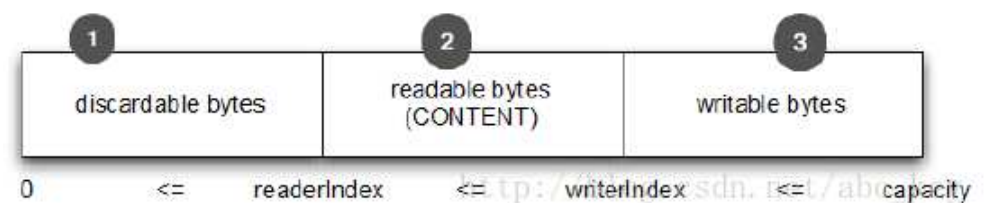
```
[java]
01. //create a ByteBuf of capacity is 16
02. ByteBuf buf = Unpooled.buffer(16);
03. //write data to buf
04. for(int i=0;i<16;i++){
05.     buf.writeByte(i+1);
06. }
07. //read data from buf
08. for(int i=0;i<buf.capacity();i++){
09.     System.out.println(buf.getByte(i));
10. }
```

注意通过索引访问时不会推进读索引和写索引，我们可以通过ByteBuf的readerIndex()或writerIndex()来分别推进读索引或写索引。

5.3.2 顺序访问索引

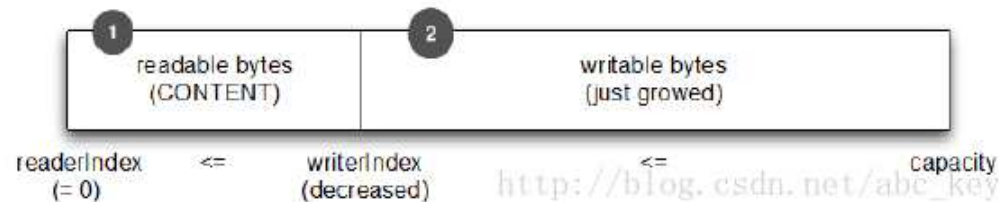
ByteBuf提供两个指针变量支付读和写操作，读操作是使用readerIndex()，写操作时使用writerIndex()。这和JDK的ByteBuffer不同，ByteBuffer只有一个方法来设置索引，所以需要使用flip()方法来切换读和写模式。

ByteBuf一定符合：0 <= readerIndex <= writerIndex <= capacity。



5.3.3 Discardable bytes废弃字节

我们可以调用ByteBuf.discardReadBytes()来回收已经读取过的字节，discardReadBytes()将丢弃从索引0到readerIndex之间的字节。调用discardReadBytes()方法后会变成如下图：



ByteBuf.discardReadBytes()可以用来清空ByteBuf中已读取的数据，从而使ByteBuf有多余的空间容纳新的数据，但是discardReadBytes()可能会涉及内存复制，因为它需要移动ByteBuf中可读的字节到开始位置，这样的操作会影响性能，一般在需要马上释放内存的时候使用收益会比较大。

5.3.4 可读字节(实际内容)

任何读取操作会增加readerIndex，如果读取操作的参数也是一个ByteBuf而没有指定目的索引，指定的目的缓冲区的writerIndex会一起增加，没有足够的内容时会抛出IndexOutOfBoundsException。新分配、包装、复制的缓冲区的readerIndex的默认值都是0。下面代码显示了获取所有可读数据：


```
[java]
01. ByteBuf buf = Unpooled.buffer(16);
02. while(buf.isReadable()){
03.     System.out.println(buf.readByte());
04. }
```

(代码于原书中有出入，原书可能是基于Netty4之前的版本讲解的，此处基于Netty4)

5.3.5 可写字节Writable bytes

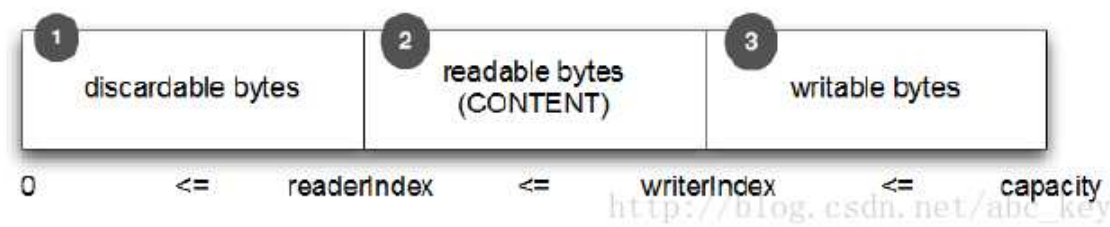
任何写的操作会增加writerIndex。若写操作的参数也是一个ByteBuf并且没有指定数据源索引，那么指定缓冲区的readerIndex也会一起增加。若没有足够的可写字节会抛出IndexOutOfBoundsException。新分配的缓冲区writerIndex的默认值是0。下面代码显示了随机一个int数字来填充缓冲区，直到缓冲区空间耗尽：

```
[java]
01. Random random = new Random();
02. ByteBuf buf = Unpooled.buffer(16);
03. while(buf.writableBytes() >= 4){
04.     buf.writeInt(random.nextInt());
05. }
```

5.3.6 清除缓冲区索引Clearing the buffer indexes

调用ByteBuf.clear()可以设置readerIndex和writerIndex为0，clear()不会清除缓冲区的内容，只是将两个索引值设置为0。请注意ByteBuf.clear()与JDK的ByteBuffer.clear()的语义不同。

下图显示了ByteBuf调用clear()之前：



下图显示了调用clear()之后：



和discardReadBytes()相比，clear()是便宜的，因为clear()不会复制任何内存。

5.3.7 搜索操作Search operations

各种indexOf()方法帮助你定位一个值的索引是否符合，我们可以用ByteBufProcessor复杂动态顺序搜索实现简单的静态单字节搜索。如果你想解码可变长度的数据，如null结尾的字符串，你会发现bytesBefore(byte value)方法有用。例如我们写一个集成的flash sockets的应用程序，这个应用程序使用NULL结束的内容，使用bytesBefore(byte value)方法可以很容易的检查数据中的空字节。没有ByteBufProcessor的话，我们需要自己做这些事情，使用ByteBufProcessor效率更好。

5.3.8 标准和重置Mark and reset

每个ByteBuf有两个标注索引，一个存储readerIndex，一个存储writerIndex。你可以通过调用一个重置方法重新定位两个索引之一，它类似于InputStream的标注和重置方法，没有读限制。我们可以通过调用readerIndex(int readerIndex)和writerIndex(int writerIndex)移动读索引和写索引到指定位置，调用这两个方法设置指定索引位置时可能抛出IndexOutOfBoundsException。

5.3.9 衍生的缓冲区Derived buffers

调用duplicate()、slice()、slice(int index, int length)、order(ByteOrder endianness)会创建一个现有缓冲区的视图。衍生的缓冲区有独立的readerIndex、writerIndex和标注索引。如果需要现有缓冲区的全新副本，可以使用copy()或copy(int index, int length)获得。看下面代码：

```
[java]
01. // get a Charset of UTF-8
02. Charset utf8 = Charset.forName("UTF-8");
03. // get a ByteBuf
04. ByteBuf buf = Unpooled.copiedBuffer("Netty in Action rocks!", utf8);
05. // slice
06. ByteBuf sliced = buf.slice(0, 14);
07. // copy
08. ByteBuf copy = buf.copy(0, 14);
09. // print "Netty in Action rocks!"
10. System.out.println(buf.toString(utf8));
11. // print "Netty in Act"
```



```
12. System.out.println(sliced.toString(utf8));
13. // print "Netty in Act"
14. System.out.println(copy.toString(utf8));
```

5.3.10 读/写操作以及其他一些操作

- 有两种主要类型的读写操作：
- get/set操作以索引为基础，在给定的索引设置或获取字节
 - 从当前索引开始读写，递增当前的写索引或读索引

ByteBuf的各种读写方法或其他一些检查方法可以看ByteBuf的源码，这里不赘述了。

5.4 ByteBufHolder

ByteBufHolder是一个辅助类，是一个接口，其实现类是DefaultByteBufHolder，还有一些实现了ByteBufHolder接口的其他接口类。ByteBufHolder的作用就是帮助更方便的访问ByteBuf中的数据，当缓冲区没用了后，可以使用这个辅助类释放资源。ByteBufHolder很简单，提供的可供访问的方法也很少。如果你想实现一个“消息对象”有效负载存储在ByteBuf，使用ByteBufHolder是一个好主意。尽管Netty提供的各种缓冲区实现类已经很容易使用，但Netty依然提供了一些使用的工具类，使得创建和使用各种缓冲区更加方便。下面会介绍一些Netty中的缓冲区工具类。

5.4.1 ByteBufAllocator

Netty支持各种ByteBuf的池实现，来使Netty提供一种称为ByteBufAllocator成为可能。ByteBufAllocator负责分配ByteBuf实例，ByteBufAllocator提供了各种分配不同ByteBuf的方法，如需要一个堆缓冲区可以使用ByteBufAllocator.heapBuffer()，需要一个直接缓冲区可以使用ByteBufAllocator.directBuffer()，需要一个复合缓冲区可以使用ByteBufAllocator.compositeBuffer()。其他方法的使用可以看ByteBufAllocator源码及注释。

获取ByteBufAllocator对象很容易，可以从Channel的alloc()获取，也可以从ChannelHandlerContext的alloc()获取。看下面代码：

```
[ java ] [ icons ] [ C ] [ P ]

01. ServerBootstrap b = new ServerBootstrap();
02. b.group(group).channel(NioServerSocketChannel.class).localAddress(new InetSocketAddress(port))
03.   .childHandler(new ChannelInitializer<SocketChannel>() {
04.       @Override
05.       protected void initChannel(SocketChannel ch) throws Exception {
06.           // get ByteBufAllocator instance by Channel.alloc()
07.           ByteBufAllocator alloc0 = ch.alloc();
08.           ch.pipeline().addLast(new ChannelInboundHandlerAdapter() {
09.               @Override
10.               public void channelActive(ChannelHandlerContext ctx) throws Exception {
11.                   //get ByteBufAllocator instance by ChannelHandlerContext.alloc()
12.                   ByteBufAllocator alloc1 = ctx.alloc();
13.                   ctx.writeAndFlush(buf.duplicate()).addListener(ChannelFutureListener.CLOSE);
14.               }
15.           });
16.       }
17.   });
```

Netty有两种不同的ByteBufAllocator实现，一个实现ByteBuf实例池将分配和回收成本以及内存使用降到最低；另一种实现是每次使用都创建一个新的ByteBuf实例。Netty默认使用PooledByteBufAllocator，我们可以通过ChannelConfig或通过引导设置一个不同的实现来改变。更多细节在后面讲述。

5.4.2 Unpooled

Unpooled也是用来创建缓冲区的工具类，Unpooled的使用也很容易。Unpooled提供了很多方法，详细方法及使用可以看API文档或Netty源码。看下面代码：

```
[ java ] [ icons ] [ C ] [ P ]

01. // 创建复合缓冲区
02. CompositeByteBuf compBuf = Unpooled.compositeBuffer();
03. // 创建堆缓冲区
04. ByteBuf heapBuf = Unpooled.buffer(8);
05. // 创建直接缓冲区
06. ByteBuf directBuf = Unpooled.directBuffer(16);
```

5.4.3 ByteBufUtil

ByteBufUtil提供了一些静态的方法，在操作ByteBuf时非常有用。ByteBufUtil提供了Unpooled之外的一些方法，也许最有价值的是hexDump(ByteBuf buffer)方法，这个方法返回指定ByteBuf中可读字节的十六进制字符串，可以用于调试程序时打印ByteBuf的内容，十六进制字符串相比字节而言对用户更友好。

5.5 Summary

本章主要学习Netty提供的缓冲区类ByteBuf的创建和简单实用以及一些操作ByteBuf的工具类。

目录	[+]
1. ChannelPipeline	
2. ChannelHandlerContext	
1. 通知下一个ChannelHandler	
2. 修改ChannelPipeline	
3. 状态模型	
4. ChannelHandler和其子类	
1. ChannelHandler中的方法	
2. ChannelInboundHandler	
3. ChannelOutboundHandler	

本章介绍

- ChannelPipeline
- ChannelHandlerContext
- ChannelHandler
- Inbound vs outbound(入站和出站)

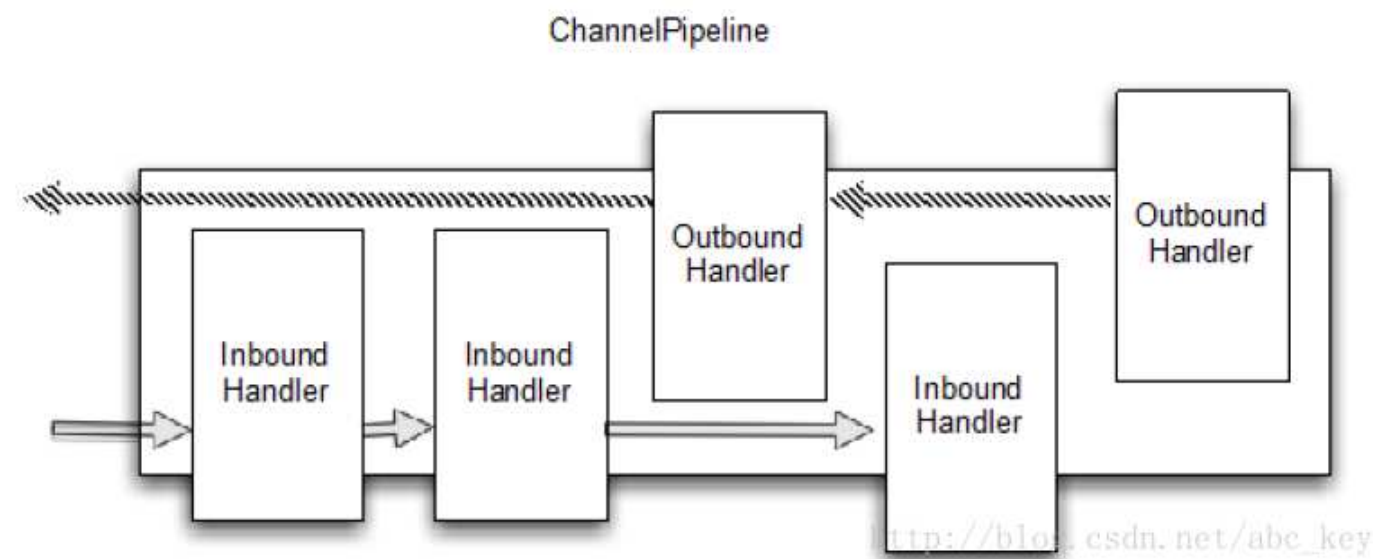
接受连接或创建他们只是你的应用程序的一部分，虽然这些任何很重要，但是一个网络应用程序旺旺是更复杂的，需要更多的代码编写，如处理传入和传出的数据。Netty提供了一个强大的处理这些事情的功能，允许用户自定义ChannelHandler的实现来处理数据。使得ChannelHandler更强大的是可以连接每个ChannelHandler来实现任务，这有助于代码的整洁和重用。但是处理数据只是ChannelHandler所做的事情之一，也可以压制I/O操作，例如写请求。所有这些都可以动态实现。

6.1 ChannelPipeline

ChannelPipeline是ChannelHandler实例的列表，用于处理或截获通道的接收和发送数据。ChannelPipeline提供了一种高级的截取过滤器模式，让用户可以在ChannelPipeline中完全控制一个事件及如何处理ChannelHandler与ChannelPipeline的交互。

对于每个新的通道，会创建一个新的ChannelPipeline并附加至通道。一旦连接，Channel和ChannelPipeline之间的耦合是永久性的。Channel不能附加其他的ChannelPipeline或从ChannelPipeline分离。

下图描述了ChannelHandler在ChannelPipeline中的I/O处理，一个I/O操作可以由一个ChannelInboundHandler或ChannelOutboundHandler进行处理，并通过调用ChannelInboundHandler处理入站IO或通过ChannelOutboundHandler处理出站IO。



如上图所示，ChannelPipeline是ChannelHandler的一个列表；如果一个入站I/O事件被触发，这个事件会从第一个开始依次通过ChannelPipeline中的ChannelHandler；若是一个入站I/O事件，则会从最后一个开始依次通过ChannelPipeline中的ChannelHandler。ChannelHandler可以处理事件并检查类型，如果某个ChannelHandler不能处理则会跳过，并将事件传递到下一个ChannelHandler。ChannelPipeline可以动态添加、删除、替换其中的ChannelHandler，这样的机制可以提高灵活性。

修改ChannelPipeline的方法：

- addFirst(...), 添加ChannelHandler在ChannelPipeline的第一个位置
- addBefore(...), 在ChannelPipeline中指定的ChannelHandler名称之前添加ChannelHandler
- addAfter(...), 在ChannelPipeline中指定的ChannelHandler名称之后添加ChannelHandler
- addLast(ChannelHandler...), 在ChannelPipeline的末尾添加ChannelHandler
- remove(...), 删除ChannelPipeline中指定的ChannelHandler
- replace(...), 替换ChannelPipeline中指定的ChannelHandler

```
[java]
01. ChannelPipeline pipeline = ch.pipeline();
02. FirstHandler firstHandler = new FirstHandler();
03. pipeline.addLast("handler1", firstHandler);
04. pipeline.addFirst("handler2", new SecondHandler());
05. pipeline.addLast("handler3", new ThirdHandler());
06. pipeline.remove("handler3");
07. pipeline.remove(firstHandler);
08. pipeline.replace("handler2", "handler4", new FourthHandler());
```

被添加到ChannelPipeline的ChannelHandler将通过IO-Thread处理事件，这意味了必须不能有其他的IO-Thread阻塞来影响IO的整体处理；有时候可能需要阻塞，例如JDBC。因此，Netty允许通过一个EventExecutorGroup到每一个ChannelPipeline.add*方法，自定义的事件会被包含在EventExecutorGroup中的EventExecutor来处理，默认的实现是DefaultEventExecutorGroup。

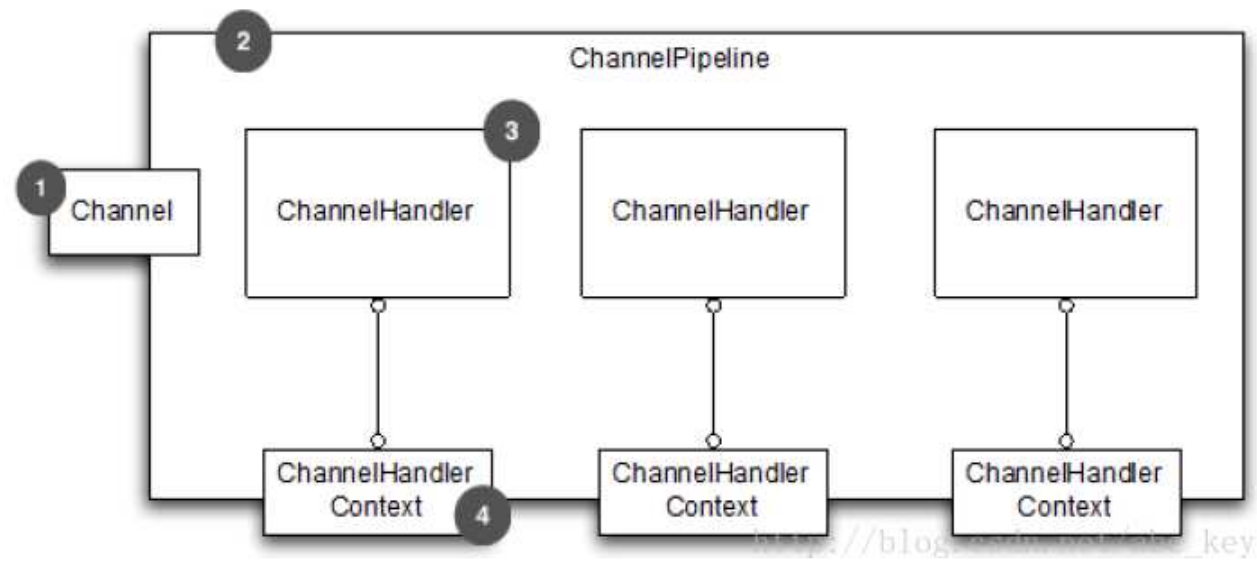
ChannelPipeline除了一些修改的方法，还有很多其他的方法，具体是方法及使用可以看API文档或源码。

6.2 ChannelHandlerContext

每个ChannelHandler被添加到ChannelPipeline后，都会创建一个ChannelHandlerContext并为之创建的ChannelHandler关联绑定。ChannelHandlerContext允许ChannelHandler与其他的ChannelHandler实现进行交互，这是相同ChannelPipeline的一部分。ChannelHandlerContext不会改变添加到其中的ChannelHandler，因此它是安全的。

6.2.1 通知下一个ChannelHandler

在相同的ChannelPipeline中通过调用ChannelInboundHandler和ChannelOutboundHandler中各个方法中的一个方法来通知最近的handler，通知开始的地方取决你如何设置。下图显示了ChannelHandlerContext、ChannelHandler、ChannelPipeline的关系：



如果你想有一些事件流全部通过ChannelPipeline，有两个不同的方法可以做到：

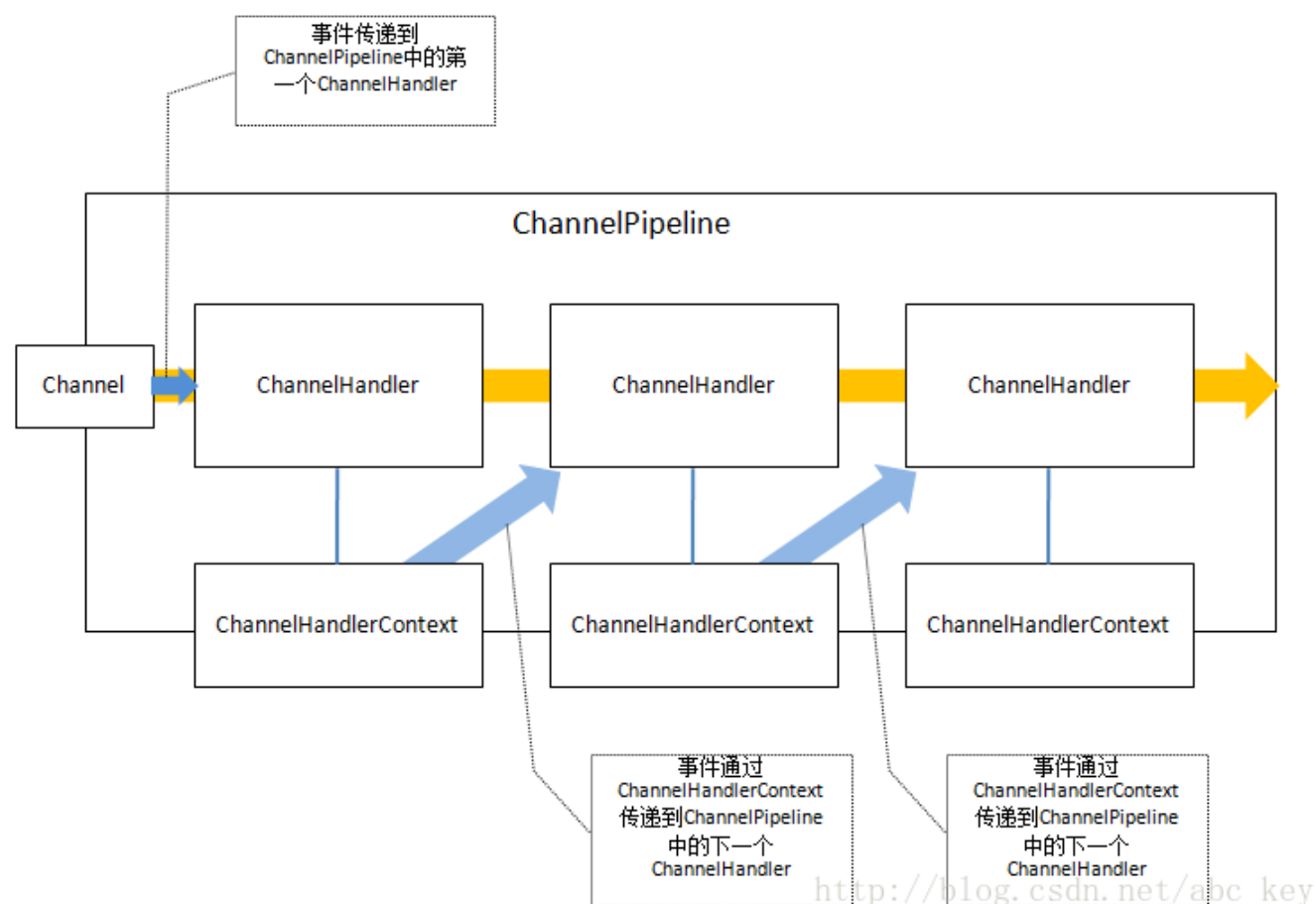
- 调用Channel的方法
- 调用ChannelPipeline的方法

这两个方法都可以让事件流全部通过ChannelPipeline。无论从头部还是尾部开始，因为它主要依赖于事件的性质。如果是一个“进站”事件，它开始于头部；若是一个“出站”事件，则开始于尾部。

下面的代码显示了一个写事件如何通过ChannelPipeline从尾部开始：

```
[java]
01. @Override
02. protected void initChannel(SocketChannel ch) throws Exception {
03.     ch.pipeline().addLast(new ChannelInboundHandlerAdapter() {
04.         @Override
05.         public void channelActive(ChannelHandlerContext ctx) throws Exception {
06.             // Event via Channel
07.             Channel channel = ctx.channel();
08.             channel.write(Unpooled.copiedBuffer("netty in action", CharsetUtil.UTF_8));
09.             // Event via ChannelPipeline
10.             ChannelPipeline pipeline = ctx.pipeline();
11.             pipeline.write(Unpooled.copiedBuffer("netty in action", CharsetUtil.UTF_8));
12.         }
13.     });
14. }
```

下图表示通过Channel或ChannelPipeline的通知：



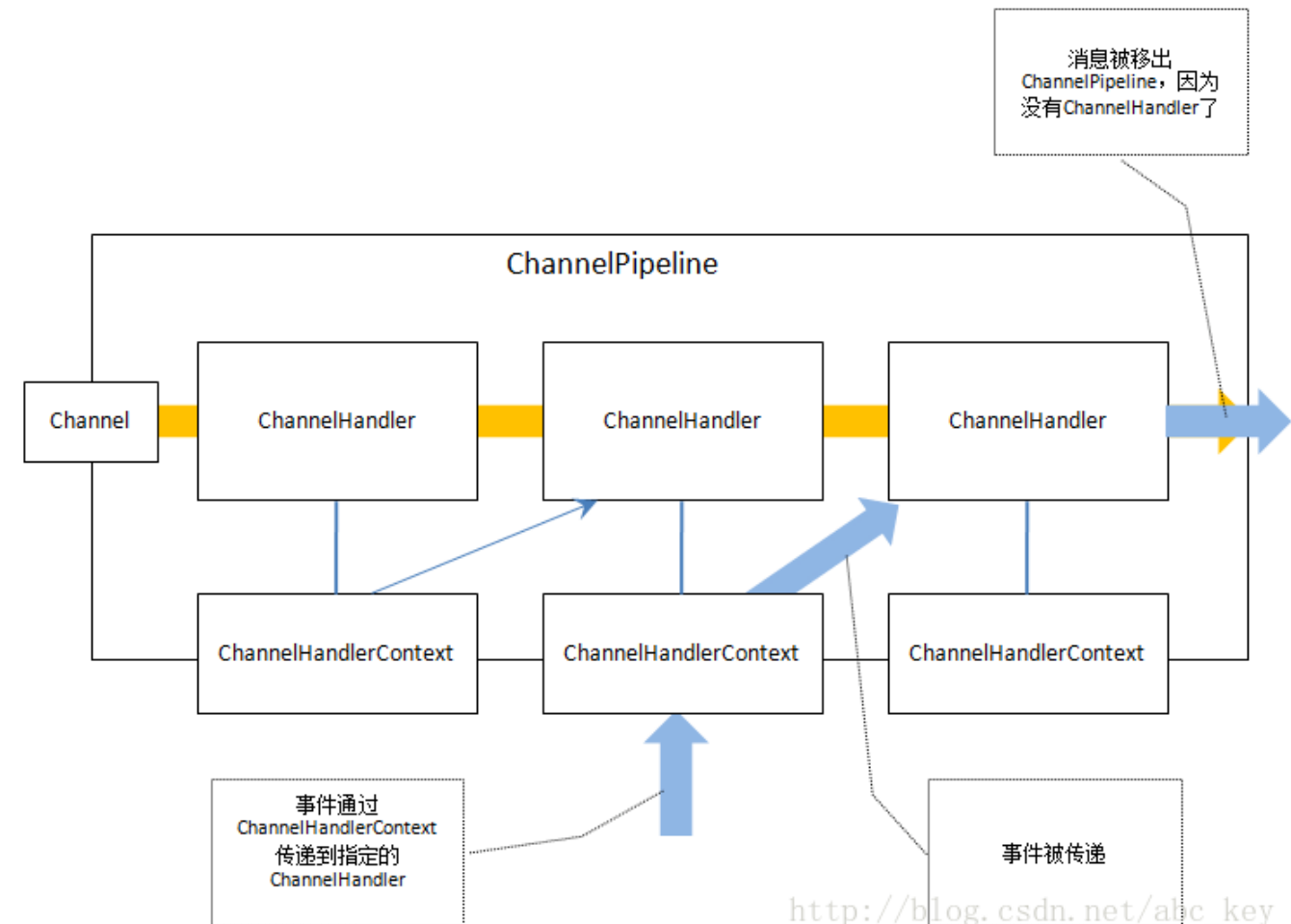
可能你想从ChannelPipeline的指定位置开始，不想流经整个ChannelPipeline，如下情况：

- 为了节省开销，不感兴趣的ChannelHandler不让通过
- 排除一些ChannelHandler

在这种情况下，你可以使用ChannelHandlerContext的ChannelHandler通知起点。它使用ChannelHandlerContext执行下一个ChannelHandler。下面代码显示了直接使用ChannelHandlerContext操作：

```
[ java]
01. // Get reference of ChannelHandlerContext
02. ChannelHandlerContext ctx = ...;
03. // Write buffer via ChannelHandlerContext
04. ctx.write(Unpooled.copiedBuffer("Netty in Action", CharsetUtil.UTF_8));
```

该消息流经ChannelPipeline到下一个ChannelHandler，在这种情况下使用ChannelHandlerContext开始下一个ChannelHandler。下图显示了事件流：



如上图显示的，从指定的ChannelHandlerContext开始，跳过前面所有的ChannelHandler，使用ChannelHandlerContext操作是常见的模式，最常用的是从ChannelHanlder调用操作，也可以在外部使用ChannelHandlerContext，因为这是线程安全的。

6.2.2 修改ChannelPipeline

调用ChannelHandlerContext的pipeline()方法能访问ChannelPipeline，能在运行时动态的增加、删除、替换ChannelPipeline中的ChannelHandler。可以保持ChannelHandlerContext供以后使用，如外部Handler方法触发一个事件，甚至从一个不同的线程。

下面代码显示了保存ChannelHandlerContext供之后使用或其他线程使用：

```
[ java ]
01. public class WiteHandler extends ChannelHandlerAdapter {
02.     private ChannelHandlerContext ctx;
03.
04.     @Override
05.     public void handlerAdded(ChannelHandlerContext ctx) throws Exception {
06.         this.ctx = ctx;
07.     }
08.
09.     public void send(String msg){
10.         ctx.write(msg);
11.     }
12. }
```

请注意，ChannelHandler实例如果带有@Sharable注解则可以被添加到多个ChannelPipeline。也就是说单个ChannelHandler实例可以有多个ChannelHandlerContext，因此可以调用不同ChannelHandlerContext获取同一个ChannelHandler。如果添加不带@Sharable注解的ChannelHandler实例到多个ChannelPipeline则会抛出异常；使用@Sharable注解后的ChannelHandler必须在不同的线程和不同的通道上安全使用。怎么是不安全的使用？看下面代码：

```
[ java ]
01. @Sharable
02. public class NotSharableHandler extends ChannelInboundHandlerAdapter {
03.
04.     private int count;
05.
06.     @Override
07.     public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {
08.         count++;
09.         System.out.println("channelRead(...) called the " + count + " time");
10.         ctx.fireChannelRead(msg);
11.     }
12.
13. }
```

上面是一个带@Sharable注解的Handler，它被多个线程使用时，里面count是不安全的，会导致count值错误。

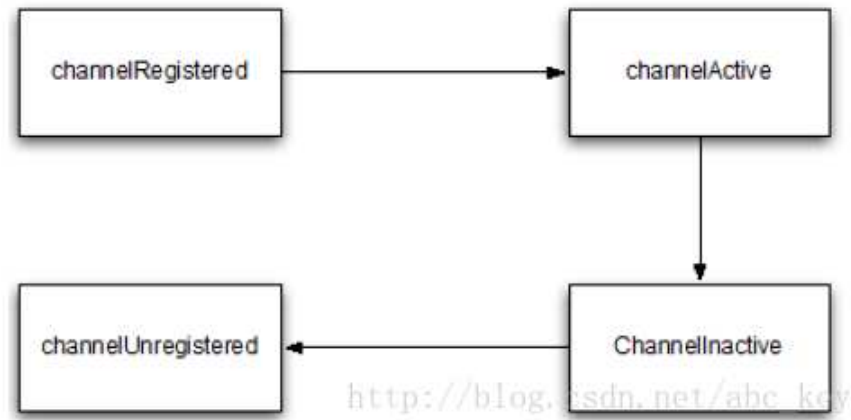
为什么要共享ChannelHandler？使用@Sharable注解共享一个ChannelHandler在一些需求中还是有很好的作用的，如使用一个ChannelHandler来统计连接数或来处理一些全局数据等等。

6.3 状态模型

Netty有一个简单但强大的状态模型，并完美映射到ChannelInboundHandler的各个方法。下面是Channel生命周期四个不同的状态：

- channelUnregistered
- channelRegistered
- channelActive
- channelInactive

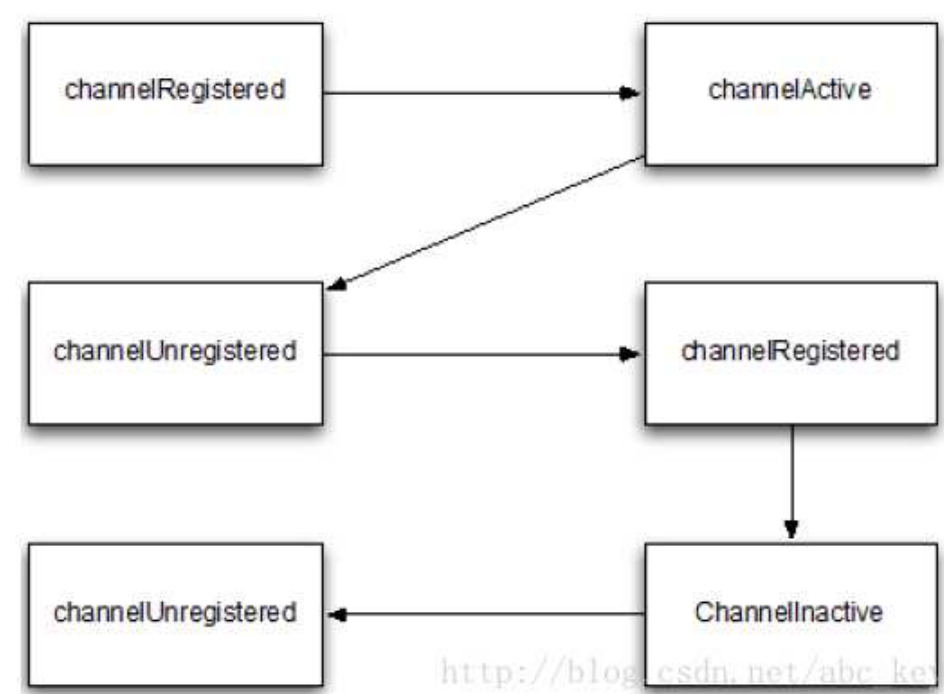
Channel的状态在其生命周期中变化，因为状态变化需要触发，下图显示了Channel状态变化：



还可以看到额外的状态变化，因为用户允许从EventLoop中注销Channel暂停事件执行，然后再重新注册。在这种情况下，你会看到

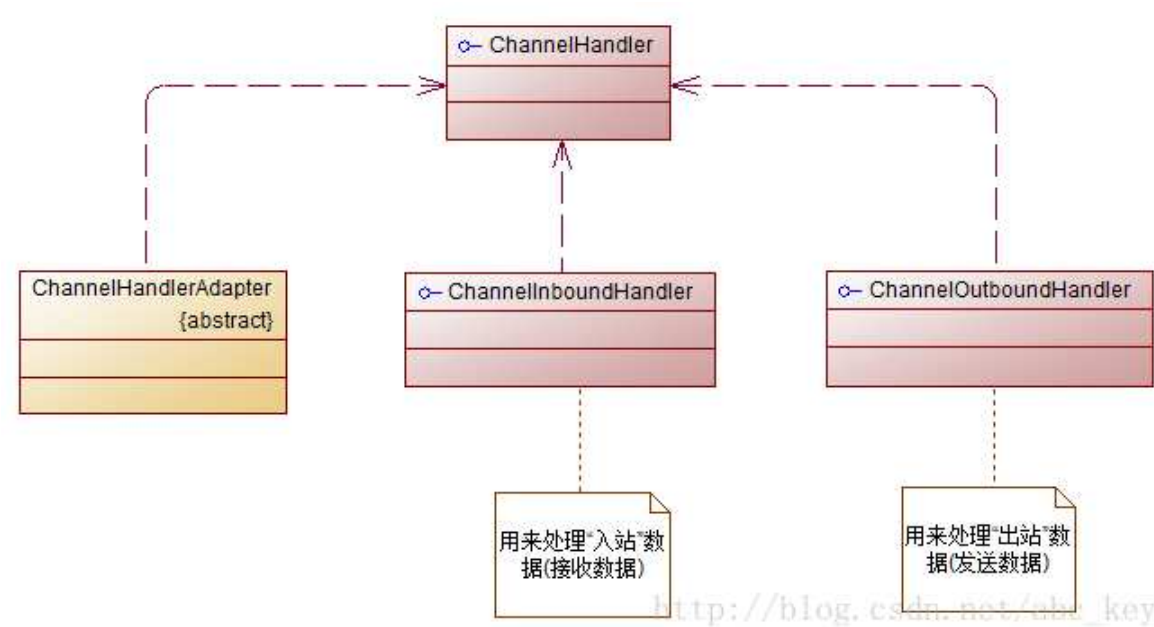
多个channelRegistered和channelUnregistered状态的变化，而永远只有一个channelActive和channellInactive的状态，因为一个通道在其生命周期内只能连接一次，之后就会被回收；重新连接，则是创建一个新的通道。

下图显示了从EventLoop中注销Channel后再重新注册的状态变化：



6.4 ChannelHandler和其子类

Netty中有3个实现了ChannelHandler接口的类，其中2个是接口，一个是抽象类。如下图：



6.4.1 ChannelHandler中的方法

Netty定义了一个良好的类型层次结构来表示不同的处理程序类型，所有的类型的父类是ChannelHandler。ChannelHandler提供了在其生命周期内添加或从ChannelPipeline中删除的方法。

- handlerAdded，ChannelHandler添加到实际上下文中准备处理事件
- handlerRemoved，将ChannelHandler从实际上下文中删除，不再处理事件
- exceptionCaught，处理抛出的异常

上面三个方法都需要传递ChannelHandlerContext参数，每个ChannelHandler被添加到ChannelPipeline时会自动创建ChannelHandlerContext。ChannelHandlerContext允许在本地通道安全的存储和检索值。Netty还提供了一个实现了ChannelHandler的抽象类：ChannelHandlerAdapter。ChannelHandlerAdapter实现了父类的所有方法，基本上就是传递事件到ChannelPipeline中的下一个ChannelHandler直到结束。

6.4.2 ChannelInboundHandler

ChannelInboundHandler提供了一些方法再接收数据或Channel状态改变时被调用。下面是ChannelInboundHandler的一些方法：

- channelRegistered，ChannelHandlerContext的Channel被注册到EventLoop；
- channelUnregistered，ChannelHandlerContext的Channel从EventLoop中注销
- channelActive，ChannelHandlerContext的Channel已激活
- channellInactive，ChannelHanderContxt的Channel结束生命周期
- channelRead，从当前Channel的对端读取消息
- channelReadComplete，消息读取完成后执行
- userEventTriggered，一个用户事件被处罚

- `channelWritabilityChanged`，改变通道的可写状态，可以使用`Channel.isWritable()`检查
- `exceptionCaught`，重写父类`ChannelHandler`的方法，处理异常

Netty提供了一个实现了`ChannelInboundHandler`接口并继承`ChannelHandlerAdapter`的类：`ChannelInboundHandlerAdapter`。`ChannelInboundHandlerAdapter`实现了`ChannelInboundHandler`的所有方法，作用就是处理消息并将消息转发到`ChannelPipeline`中的下一个`ChannelHandler`。`ChannelInboundHandlerAdapter`的`channelRead`方法处理完消息后不会自动释放消息，若想自动释放收到的消息，可以使用`SimpleChannelInboundHandler<I>`。

看下面代码：

```
[ java ]
01.  /**
02.   * 实现ChannelInboundHandlerAdapter的Handler，不会自动释放接收的消息对象
03.   * @author c.k
04.   *
05.   */
06.  public class DiscardHandler extends ChannelInboundHandlerAdapter {
07.      @Override
08.      public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {
09.          // 手动释放消息
10.          ReferenceCountUtil.release(msg);
11.      }
12.  }
```

```
[ java ]
01.  /**
02.   * 继承SimpleChannelInboundHandler，会自动释放消息对象
03.   * @author c.k
04.   *
05.   */
06.  public class SimpleDiscardHandler extends SimpleChannelInboundHandler<Object> {
07.      @Override
08.      protected void channelRead0(ChannelHandlerContext ctx, Object msg) throws Exception {
09.          // 不需要手动释放
10.      }
11.  }
```

如果需要其他状态改变的通知，可以重写`Handler`的其他方法。通常自定义消息类型来解码字节，可以实现`ChannelInboundHandler`或`ChannelInboundHandlerAdapter`。有一个更好的解决方法，使用编解码器的框架可以很容的实现。使用`ChannelInboundHandler`、`ChannelInboundHandlerAdapter`、`SimpleChannelInboundhandler`这三个中的一个来处理接收消息，使用哪一个取决于需求；大多数时候使用`SimpleChannelInboundHandler`处理消息，使用`ChannelInboundHandlerAdapter`处理其他的“入站”事件或状态改变。

`ChannelInitializer`用来初始化`ChannelHandler`，将自定义的各种`ChannelHandler`添加到`ChannelPipeline`中。

6.4.3 ChannelOutboundHandler

`ChannelOutboundHandler`用来处理“出站”的数据消息。`ChannelOutboundHandler`提供了下面一些方法：

- `bind`，`Channel`绑定本地地址
- `connect`，`Channel`连接操作
- `disconnect`，`Channel`断开连接
- `close`，关闭`Channel`
- `deregister`，注销`Channel`
- `read`，读取消息，实际是截获`ChannelHandlerContext.read()`
- `write`，写操作，实际是通过`ChannelPipeline`写消息，`Channel.flush()`属性到实际通道
- `flush`，刷新消息到通道

`ChannelOutboundHandler`是`ChannelHandler`的子类，实现了`ChannelHandler`的所有方法。所有最重要的方法采取`ChannelPromise`，因此一旦请求停止从`ChannelPipeline`转发参数则必须得到通知。Netty提供了`ChannelOutboundHandler`的实现：`ChannelOutboundHandlerAdapter`。`ChannelOutboundHandlerAdapter`实现了父类的所有方法，并且可以根据需要重写感兴趣的方法。所有这些方法的实现，在默认情况下，都是通过调用`ChannelHandlerContext`的方法将事件转发到`ChannelPipeline`中下一个`ChannelHandler`。

看下面的代码：

```
[ java ]
01.  public class DiscardOutboundHandler extends ChannelOutboundHandlerAdapter {
02.      @Override
03.      public void write(ChannelHandlerContext ctx, Object msg, ChannelPromise promise) throws Exception {
```

```
04.         ReferenceCountUtil.release(msg);
05.         promise.setSuccess();
06.     }
07. }
```

重要的是要记得释放致远并直通ChannelPromise，若ChannelPromise没有被通知可能会导致其中一个ChannelFutureListener不被通知去处理一个消息。

如果消息被消费并且没有被传递到ChannelPipeline中的下一个ChannelOutboundHandler，那么就需要调用ReferenceCountUtil.release(message)来释放消息资源。一旦消息被传递到实际的通道，它会自动写入消息或在通道关闭是释放。

Netty In Action中文版 - 第七章：编解码器Codec

目录

[~]

1. 编解码器Codec

2. 解码器

1. ByteToMessageDecoder

2. ReplayingDecoder

3. MessageToMessageDecoder

4. 解码器总结

3. 编码器

1. MessageToByteEncoder

2. MessageToMessageEncoder

4. 编解码器

1. byte-to-byte编解码器

2. ByteToMessageCodec

3. MessageToMessageCodec

5. 其他编解码方式

1. CombinedChannelDuplexHandler

本章介绍

- Codec, 编解码器
- Decoder, 解码器
- Encoder, 编码器

Netty提供了编解码器框架，使得编写自定义的编解码器很容易，并且也很容易重用和封装。本章讨论Netty的编解码器框架以及使用。

7.1 编解码器Codec

编写一个网络应用程序需要实现某种编解码器，编解码器的作用就是讲原始字节数据与自定义的消息对象进行互转。网络中都是以字节码的数据形式来传输数据的，服务器编码数据后发送到客户端，客户端需要对数据进行解码，因为编解码器由两部分组成：

- Decoder(解码器)
- Encoder(编码器)

解码器负责将消息从字节或其他序列形式转成指定的消息对象，编码器则相反：解码器负责处理“进站”数据，编码器负责处理“出站”数据。编码器和解码器的结构很简单，消息被编码后解码后会自动通过`ReferenceCountUtil.release(message)`释放，如果不想释放消息可以使用`ReferenceCountUtil.retain(message)`，这将会使引用数量增加而没有消息发布，大多数时候不需要这么做。

7.2 解码器

Netty提供了丰富的解码器抽象基类，我们可以很容易的实现这些基类来自定义解码器。下面是解码器的一个类型：

- 解码字节到消息
- 解码消息到消息
- 解码消息到字节

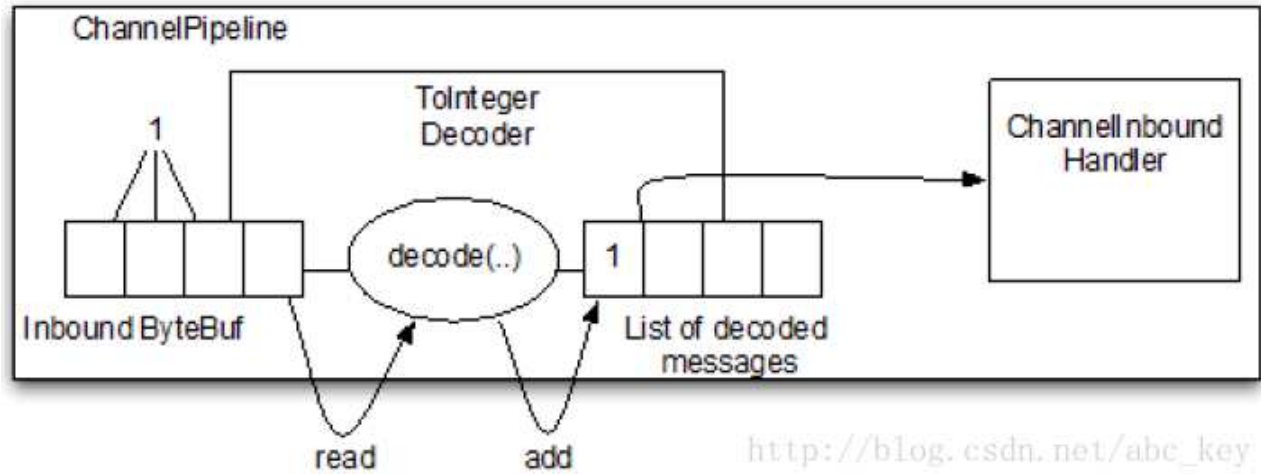
本章将概述不同的抽象基类，来帮助了解解码器的实现。深入了解Netty提供的解码器之前先了解解码器的作用是什么？解码器负责解码“入站”数据从一种格式到另一种格式，解码器处理入站数据是抽象ChannelInboundHandler的实现。实践中使用解码器很简单，就是将入站数据转换格式后传递到ChannelPipeline中的下一个ChannelInboundHandler进行处理；这样的处理时很灵活的，我们可以将解码器放在ChannelPipeline中，重用逻辑。

7.2.1 ByteToMessageDecoder

通常你需要将消息从字节解码成消息或者从字节解码成其他的序列化字节。这是一个常见的任务，Netty提供了抽象基类，我们可以使用它们来实现。Netty中提供的ByteToMessageDecoder可以将字节消息解码成POJO对象，下面列出了ByteToMessageDecoder两个主要方法：

- `decode(ChannelHandlerContext, ByteBuf, List<Object>)`, 这个方法是唯一的一个需要自己实现的抽象方法，作用是将`ByteBuf`数据解码成其他形式的数据。
- `decodeLast(ChannelHandlerContext, ByteBuf, List<Object>)`, 实际上调用的是`decode(...)`。

例如服务器从某个客户端接收到一个整数值的字节码，服务器将数据读入ByteBuf并经过ChannelPipeline中的每个ChannelInboundHandler进行处理，看下图：



上图显示了从“进站”ByteBuffer读取bytes后由ToIntegerDecoder进行解码，然后向解码后的消息传递到ChannelPipeline中的下一个ChannelInboundHandler。看下面ToIntegerDecoder的实现代码：

```
[java]
01. /**
02.  * Integer 解码器, ByteToMessageDecoder 实现
03.  * @author c.k
04.  *
05.  */
06. public class ToIntegerDecoder extends ByteToMessageDecoder {
07.
08.     @Override
09.     protected void decode(ChannelHandlerContext ctx, ByteBuffer in, List<Object> out) throws Exception {
10.         if(in.readableBytes() >= 4){
11.             out.add(in.readInt());
12.         }
13.     }
14. }
```

从上面的代码可能会发现，我们需要检查ByteBuffer读之前是否有足够的字节，若没有这个检查岂不更好？是的，Netty提供了这样的处理允许byte-to-message解码,在下一节讲解。除了ByteToMessageDecoder之外，Netty还提供了许多其他的解码接口。

7.2.2 ReplayingDecoder

ReplayingDecoder是byte-to-message解码的一种特殊的抽象基类，读取缓冲区的数据之前需要检查缓冲区是否有足够的字节，使用ReplayingDecoder就无需自己检查；若ByteBuffer中有足够的字节，则会正常读取；若没有足够的字节则会停止解码。也正因为这样的包装使得ReplayingDecoder带有一定的局限性。

- 不是所有的操作都被ByteBuffer支持，如果调用一个不支持的操作会抛出DecoderException。
- ByteBuffer.readableBytes()大部分时间不会返回期望值

如果你能忍受上面列出的限制，相比ByteToMessageDecoder，你可能更喜欢ReplayingDecoder。在满足需求的情况下推荐使用ByteToMessageDecoder，因为它的处理比较简单，没有ReplayingDecoder实现的那么复杂。ReplayingDecoder继承与ByteToMessageDecoder，所以他们提供的接口是相同的。下面代码是ReplayingDecoder的实现：

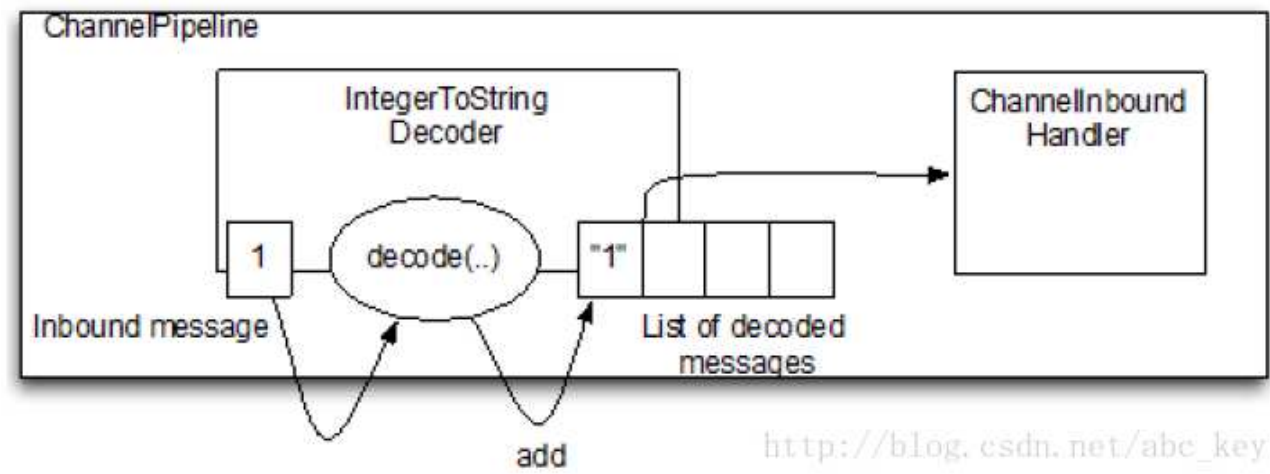
```
[java]
01. /**
02.  * Integer 解码器, ReplayingDecoder 实现
03.  * @author c.k
04.  *
05.  */
06. public class ToIntegerReplayingDecoder extends ReplayingDecoder<Void> {
07.
08.     @Override
09.     protected void decode(ChannelHandlerContext ctx, ByteBuffer in, List<Object> out) throws Exception {
10.         out.add(in.readInt());
11.     }
12. }
```

当从接收的数据ByteBuffer读取integer，若没有足够的字节可读，decode(..)会停止解码，若有足够的字节可读，则会读取数据添加到List列表中。使用ReplayingDecoder或ByteToMessageDecoder是个人喜好的问题，Netty提供了这两种实现，选择哪一个都可以。

上面讲了byte-to-message的解码实现方式，那message-to-message该如何实现呢？Netty提供了MessageToMessageDecoder抽象类。

7.2.3 MessageToMessageDecoder

将消息对象转成消息对象可是使用MessageToMessageDecoder，它是一个抽象类，需要我们自己实现其decode(...)。message-to-message同上面讲的byte-to-message的处理机制一样，看下图：



看下面的实现代码：

```
[ java ]  
01.  /**  
02.  * 将接收的Integer消息转成String类型，MessageToMessageDecoder实现  
03.  * @author c.k  
04.  *  
05.  */  
06.  public class IntegerToStringDecoder extends MessageToMessageDecoder<Integer> {  
07.  
08.      @Override  
09.      protected void decode(ChannelHandlerContext ctx, Integer msg, List<Object> out) throws Exception {  
10.          out.add(String.valueOf(msg));  
11.      }  
12.  }
```

7.2.4 解码器总结

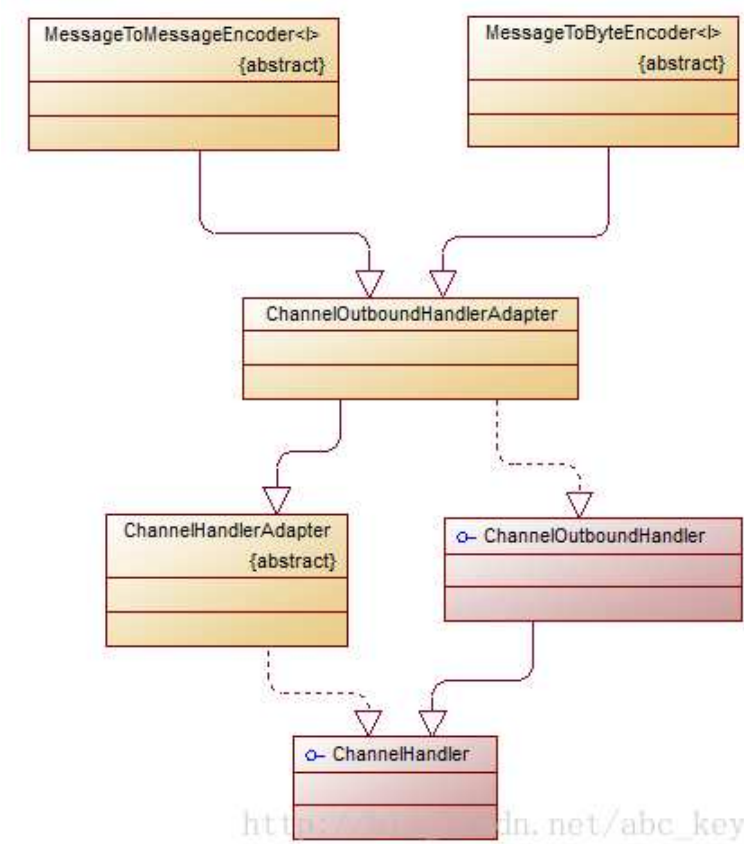
解码器是用来处理入站数据，Netty提供了很多解码器的实现，可以根据需求详细了解。那我们发送数据需要将数据编码，Netty中也提供了编码器的支持。下一节将讲解如何实现编码器。

7.3 编码器

Netty提供了一些基类，我们可以很简单的编码器。同样的，编码器有下面两种类型：

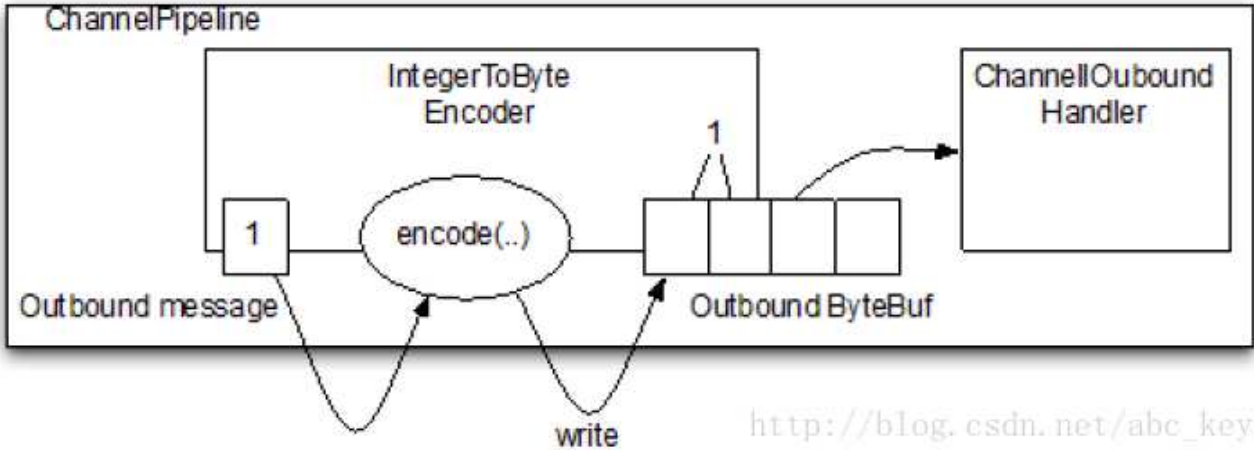
- 消息对象编码成消息对象
- 消息对象编码成字节码

相对解码器，编码器少了一个byte-to-byte的类型，因为出站数据这样做没有意义。编码器的作用就是将处理好的数据转成字节码以便在网络中传输。对照上面列出的两种编码器类型，Netty也分别提供了两个抽象类：MessageToByteEncoder和MessageToMessageEncoder。下面是类关系图：



7.3.1 MessageToByteEncoder

MessageToByteEncoder是抽象类，我们自定义一个继承MessageToByteEncoder的编码器只需要实现其提供的encode(...)方法。其工作流程如下图：

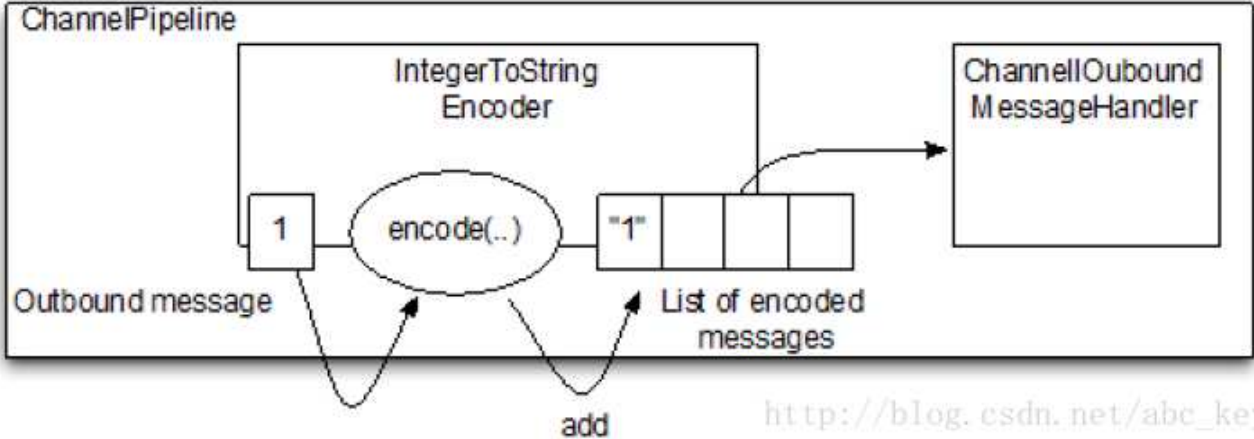


实现代码如下：

```
[java]
01. /**
02.  * 编码器，将Integer值编码成byte[]，MessageToByteEncoder实现
03.  * @author c.k
04.  *
05.  */
06. public class IntegerToByteEncoder extends MessageToByteEncoder<Integer> {
07.     @Override
08.     protected void encode(ChannelHandlerContext ctx, Integer msg, ByteBuf out) throws Exception {
09.         out.writeInt(msg);
10.     }
11. }
```

7.3.2 MessageToMessageEncoder

需要将消息编码成其他的消息时可以使用Netty提供的MessageToMessageEncoder抽象类来实现。例如将Integer编码成String，其工作流程如下图：



代码实现如下：

```
[java]
01. /**
02.  * 编码器，将Integer编码成String，MessageToMessageEncoder实现
03.  * @author c.k
04.  *
05.  */
06. public class IntegerToStringEncoder extends MessageToMessageEncoder<Integer> {
07.     @Override
08.     protected void encode(ChannelHandlerContext ctx, Integer msg, List<Object> out) throws Exception {
09.         out.add(String.valueOf(msg));
10.     }
11. }
12. }
```

7.4 编解码器

实际编码中，一般会将编码和解码操作封装太一个类中，解码处理“入站”数据，编码处理“出站”数据。知道了编码和解码器，对于下面的情况不会感觉惊讶：

- byte-to-message编码和解码
- message-to-message编码和解码

如果确定需要在ChannelPipeline中使用编码器和解码器，需要更好的使用一个抽象的编解码器。同样，使用编解码器的时候，不可能只删除解码器或编码器而离开ChannelPipeline导致某种不一致的状态。使用编解码器将强制性的要么都在ChannelPipeline，要么都不

在ChannelPipeline。

考虑到这一点，我们在下面几节将更深入的分析Netty提供的编解码抽象类。

7.4.1 byte-to-byte编解码器

Netty4较之前的版本，其结构有很大的变化，在Netty4中实现byte-to-byte提供了2个类：ByteArrayEncoder和ByteArrayDecoder。这两个类用来处理字节到字节的编码和解码。下面是这两个类的源码，一看就知道是如何处理的：

```
[ java ]
01. public class ByteArrayDecoder extends MessageToMessageDecoder<ByteBuf> {
02.     @Override
03.     protected void decode(ChannelHandlerContext ctx, ByteBuf msg, List<Object> out) throws Exception {
04.         // copy the ByteBuf content to a byte array
05.         byte[] array = new byte[msg.readableBytes()];
06.         msg.getBytes(0, array);
07.
08.         out.add(array);
09.     }
10. }
```

```
[ java ]
01. @Sharable
02. public class ByteArrayEncoder extends MessageToMessageEncoder<byte[]> {
03.     @Override
04.     protected void encode(ChannelHandlerContext ctx, byte[] msg, List<Object> out) throws Exception {
05.         out.add(Unpooled.wrappedBuffer(msg));
06.     }
07. }
```

7.4.2 ByteToMessageCodec

ByteToMessageCodec用来处理byte-to-message和message-to-byte。如果想要解码字节消息成POJO或编码POJO消息成字节，对于这种情况，ByteToMessageCodec<I>是一个不错的选择。ByteToMessageCodec是一种组合，其等同于ByteToMessageDecoder和MessageToByteEncoder的组合。MessageToByteEncoder是个抽象类，其中有2个方法需要我们自己实现：

- encode(ChannelHandlerContext, I, ByteBuf)，编码
- decode(ChannelHandlerContext, ByteBuf, List<Object>)，解码

7.4.3 MessageToMessageCodec

MessageToMessageCodec用于message-to-message的编码和解码，可以看成是MessageToMessageDecoder和MessageToMessageEncoder的组合体。MessageToMessageCodec是抽象类，其中有2个方法需要我们自己实现：

- encode(ChannelHandlerContext, OUTBOUND_IN, List<Object>)
- decode(ChannelHandlerContext, INBOUND_IN, List<Object>)

但是，这种编解码器能有用吗？

有许多用例，最常见的就是需要将消息从一个API转到另一个API。这种情况下需要自定义API或旧的API使用另一种消息类型。下面的代码显示了在WebSocket框架APIs之间转换消息：

```
[ java ]
01. package netty.in.action;
02.
03. import java.util.List;
04.
05. import io.netty.buffer.ByteBuf;
06. import io.netty.channel.ChannelHandlerContext;
07. import io.netty.channel.ChannelHandler.Sharable;
08. import io.netty.handler.codec.MessageToMessageCodec;
09. import io.netty.handler.codec.http.websocketx.BinaryWebSocketFrame;
10. import io.netty.handler.codec.http.websocketx.CloseWebSocketFrame;
11. import io.netty.handler.codec.http.websocketx.ContinuationWebSocketFrame;
12. import io.netty.handler.codec.http.websocketx.PingWebSocketFrame;
13. import io.netty.handler.codec.http.websocketx.PongWebSocketFrame;
14. import io.netty.handler.codec.http.websocketx.TextWebSocketFrame;
15. import io.netty.handler.codec.http.websocketx.WebSocketFrame;
16.
17. @Sharable
18. public class WebSocketConvertHandler extends
19.     MessageToMessageCodec<WebSocketFrame, WebSocketConvertHandler.MyWebSocketFrame> {
20.
21.     public static final WebSocketConvertHandler INSTANCE = new WebSocketConvertHandler();
22. }
```

```
23. @Override
24. protected void encode(ChannelHandlerContext ctx, MyWebSocketFrame msg, List<Object> out) throws Exception {
25.     switch (msg.getType()) {
26.     case BINARY:
27.         out.add(new BinaryWebSocketFrame(msg.getData()));
28.         break;
29.     case CLOSE:
30.         out.add(new CloseWebSocketFrame(true, 0, msg.getData()));
31.         break;
32.     case PING:
33.         out.add(new PingWebSocketFrame(msg.getData()));
34.         break;
35.     case PONG:
36.         out.add(new PongWebSocketFrame(msg.getData()));
37.         break;
38.     case TEXT:
39.         out.add(new TextWebSocketFrame(msg.getData()));
40.         break;
41.     case CONTINUATION:
42.         out.add(new ContinuationWebSocketFrame(msg.getData()));
43.         break;
44.     default:
45.         throw new IllegalStateException("Unsupported websocket msg " + msg);
46.     }
47. }
48.
49. @Override
50. protected void decode(ChannelHandlerContext ctx, WebSocketFrame msg, List<Object> out) throws Exception {
51.     if (msg instanceof BinaryWebSocketFrame) {
52.         out.add(new MyWebSocketFrame(MyWebSocketFrame.FrameType.BINARY, msg.content().copy()));
53.         return;
54.     }
55.     if (msg instanceof CloseWebSocketFrame) {
56.         out.add(new MyWebSocketFrame(MyWebSocketFrame.FrameType.CLOSE, msg.content().copy()));
57.         return;
58.     }
59.     if (msg instanceof PingWebSocketFrame) {
60.         out.add(new MyWebSocketFrame(MyWebSocketFrame.FrameType.PING, msg.content().copy()));
61.         return;
62.     }
63.     if (msg instanceof PongWebSocketFrame) {
64.         out.add(new MyWebSocketFrame(MyWebSocketFrame.FrameType.PONG, msg.content().copy()));
65.         return;
66.     }
67.     if (msg instanceof TextWebSocketFrame) {
68.         out.add(new MyWebSocketFrame(MyWebSocketFrame.FrameType.TEXT, msg.content().copy()));
69.         return;
70.     }
71.     if (msg instanceof ContinuationWebSocketFrame) {
72.         out.add(new MyWebSocketFrame(MyWebSocketFrame.FrameType.CONTINUATION, msg.content().copy()));
73.         return;
74.     }
75.     throw new IllegalStateException("Unsupported websocket msg " + msg);
76. }
77.
78. public static final class MyWebSocketFrame {
79.     public enum FrameType {
80.         BINARY, CLOSE, PING, PONG, TEXT, CONTINUATION
81.     }
82.
83.     private final FrameType type;
84.     private final ByteBuf data;
85.
86.     public MyWebSocketFrame(FrameType type, ByteBuf data) {
87.         this.type = type;
88.         this.data = data;
89.     }
90.
91.     public FrameType getType() {
92.         return type;
93.     }
94.
95.     public ByteBuf getData() {
96.         return data;
97.     }
98.
99. }
100. }
```


7.5 其他编解码方式

使用编解码器来充当编码器和解码器的组合失去了单独使用编码器或解码器的灵活性，编解码器是要么都有要么都没有。你可能想知道是否有解决这个僵化问题的方式，还可以让编码器和解码器在ChannelPipeline中作为一个逻辑单元。幸运的是，Netty提供了一种解决方案，使用CombinedChannelDuplexHandler。虽然这个类不是编解码器API的一部分，但是它经常被用来简历一个编解码器。

7.5.1 CombinedChannelDuplexHandler

如何使用CombinedChannelDuplexHandler来结合解码器和编码器呢？下面我们从两个简单的例子看了解。

```
[ java ]
01.  /**
02.   * 解码器，将byte转成char
03.   * @author c.k
04.   *
05.   */
06. public class ByteToCharDecoder extends ByteToMessageDecoder {
07.
08.     @Override
09.     protected void decode(ChannelHandlerContext ctx, ByteBuf in, List<Object> out) throws Exception {
10.         while(in.readableBytes() >= 2){
11.             out.add(Character.valueOf(in.readChar()));
12.         }
13.     }
14.
15. }
```

```
[ java ]
01.  /**
02.   * 编码器，将char转成byte
03.   * @author Administrator
04.   *
05.   */
06. public class CharToByteEncoder extends MessageToByteEncoder<Character> {
07.
08.     @Override
09.     protected void encode(ChannelHandlerContext ctx, Character msg, ByteBuf out) throws Exception {
10.         out.writeChar(msg);
11.     }
12. }
```

```
[ java ]
01.  /**
02.   * 继承CombinedChannelDuplexHandler，用于绑定解码器和编码器
03.   * @author c.k
04.   *
05.   */
06. public class CharCodec extends CombinedChannelDuplexHandler<ByteToCharDecoder, CharToByteEncoder> {
07.     public CharCodec(){
08.         super(new ByteToCharDecoder(), new CharToByteEncoder());
09.     }
10. }
```

从上面代码可以看出，使用CombinedChannelDuplexHandler绑定解码器和编码器很容易实现，比使用*Codec更灵活。Netty还提供了其他的协议支持，放在io.netty.handler.codec包下，如：

- Google的protobuf，在io.netty.handler.codec.protobuf包下
- Google的SPDY协议
- RTSP(Real Time Streaming Protocol，实时流传输协议)，在io.netty.handler.codec.rtsp包下
- SCTP(Stream Control Transmission Protocol，流控制传输协议)，在io.netty.handler.codec.sctp包下
-


```
10.         this.startTls = startTls;
11.     }
12.
13.     @Override
14.     protected void initChannel(Channel ch) throws Exception {
15.         SSLEngine engine = context.createSSLEngine();
16.         engine.setUseClientMode(client);
17.         ch.pipeline().addFirst("ssl", new SslHandler(engine, startTls));
18.     }
19. }
```

需要注意一点，SslHandler必须要添加到ChannelPipeline的第一个位置，可能有一些例外，但是最好这样做。回想一下之前讲解的ChannelHandler，ChannelPipeline就像是一个在处理“入站”数据时先进先出，在处理“出站”数据时后进先出的队列。最先添加的SslHandler会啊在其他Handler处理逻辑数据之前对数据进行加密，从而确保Netty服务端的所有的Handler的变化都是安全的。

SslHandler提供了一些有用的方法，可以用来修改其行为或得到通知，一旦SSL/TLS完成握手(在握手过程中的两个对等通道互相验证对方，然后选择一个加密码密)，SSL/TLS是自动执行的。看下面方法列表：

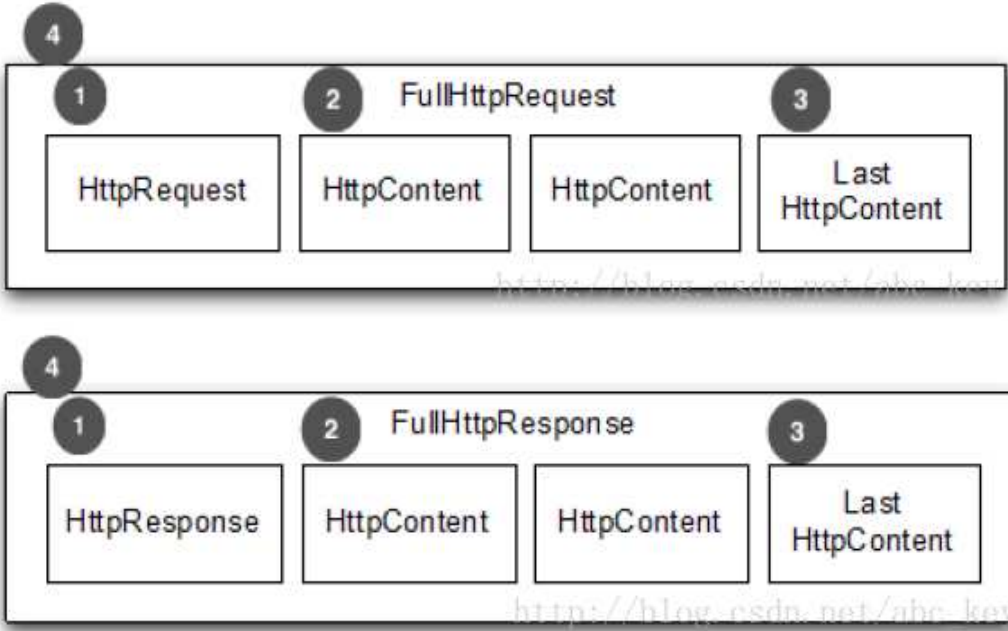
- setHandshakeTimeout(long handshakeTimeout, TimeUnit unit)，设置握手超时时间，ChannelFuture将得到通知
- setHandshakeTimeoutMillis(long handshakeTimeoutMillis)，设置握手超时时间，ChannelFuture将得到通知
- getHandshakeTimeoutMillis()，获取握手超时时间值
- setCloseNotifyTimeout(long closeNotifyTimeout, TimeUnit unit)，设置关闭通知超时时间，若超时，ChannelFuture会关闭失败
- setHandshakeTimeoutMillis(long handshakeTimeoutMillis)，设置关闭通知超时时间，若超时，ChannelFuture会关闭失败
- getCloseNotifyTimeoutMillis()，获取关闭通知超时时间
- handshakeFuture()，返回完成握手后的ChannelFuture
- close()，发送关闭通知请求关闭和销毁

8.2 使用Netty创建HTTP/HTTPS程序

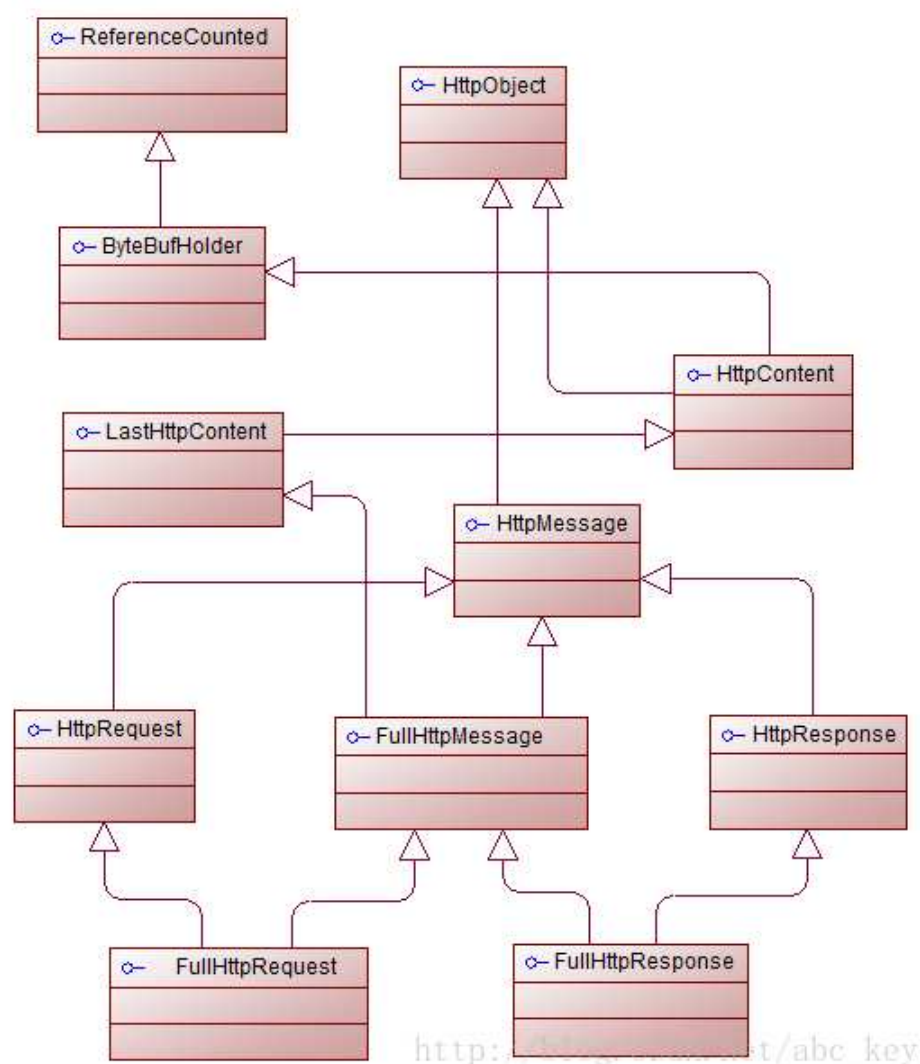
HTTP/HTTPS是最常用的协议之一，可以通过HTTP/HTTPS访问网站，或者是提供对外公开的接口服务等等。Netty附带了使用HTTP/HTTPS的handlers，而不需要我们来编写编解码器。

8.2.1 Netty的HTTP编码器，解码器和编解码器

HTTP是请求-响应模式，客户端发送一个http请求，服务就响应此请求。Netty提供了简单的编码解码HTTP协议消息的Handler。下图显示了http请求和响应：



如上面两个图所示，一个HTTP请求/响应消息可能包含不止一个，但最终都会有LastHttpContent消息。FullHttpRequest和FullHttpResponse是Netty提供的两个接口，分别用来完成http请求和响应。所有的HTTP消息类型都实现了HttpObject接口。下面是类关系图：



Netty提供了HTTP请求和响应的编码器和解码器，看下面列表：

- HttpRequestEncoder，将HttpRequest或HttpContent编码成ByteBuf
- HttpRequestDecoder，将ByteBuf解码成HttpRequest和HttpContent
- HttpResponseEncoder，将HttpResponse或HttpContent编码成ByteBuf
- HttpResponseDecoder，将ByteBuf解码成HttpResponse和HttpContent

看下面代码：

```
[java]
01. public class HttpDecoderEncoderInitializer extends ChannelInitializer<Channel> {
02.
03.     private final boolean client;
04.
05.     public HttpDecoderEncoderInitializer(boolean client) {
06.         this.client = client;
07.     }
08.
09.     @Override
10.     protected void initChannel(Channel ch) throws Exception {
11.         ChannelPipeline pipeline = ch.pipeline();
12.         if (client) {
13.             pipeline.addLast("decoder", new HttpResponseDecoder());
14.             pipeline.addLast("encoder", new HttpRequestEncoder());
15.         } else {
16.             pipeline.addLast("decoder", new HttpRequestDecoder());
17.             pipeline.addLast("encoder", new HttpResponseEncoder());
18.         }
19.     }
20. }
```

如果你需要在ChannelPipeline中有一个解码器和编码器，还分别有一个在客户端和服务端简单的编解码器：HttpClientCodec和HttpServerCodec。

在ChannelPipeline中有解码器和编码器(或编解码器)后就可以操作不同的HttpObject消息了；但是HTTP请求和响应可以有多个消息数据，你需要处理不同的部分，可能也需要聚合这些消息数据，这是很麻烦的。为了解决这个问题，Netty提供了一个聚合器，它将消息部分合并到FullHttpRequest和FullHttpResponse，因此不需要担心接收碎片消息数据。

8.2.2 HTTP消息聚合

处理HTTP时可能接收HTTP消息片段，Netty需要缓冲直到接收完整消息。要完成的处理HTTP消息，并且内存开销也不会很大，Netty为此提供了HttpObjectAggregator。通过HttpObjectAggregator，Netty可以聚合HTTP消息，使用FullHttpResponse和FullHttpRequest到

ChannelPipeline中的下一个ChannelHandler，这就消除了断裂消息，保证了消息的完整。下面代码显示了如何聚合：




```
[java]    

01.  /**
02.   * 添加聚合http消息的Handler
03.   *
04.   * @author c.k
05.   *
06.   */
07. public class HttpAggregatorInitializer extends ChannelInitializer<Channel> {
08.
09.     private final boolean client;
10.
11.     public HttpAggregatorInitializer(boolean client) {
12.         this.client = client;
13.     }
14.
15.     @Override
16.     protected void initChannel(Channel ch) throws Exception {
17.         ChannelPipeline pipeline = ch.pipeline();
18.         if (client) {
19.             pipeline.addLast("codec", new HttpClientCodec());
20.         } else {
21.             pipeline.addLast("codec", new HttpServerCodec());
22.         }
23.         pipeline.addLast("aggregator", new HttpObjectAggregator(512 * 1024));
24.     }
25.
26. }
```

如上面代码，很容使用Netty自动聚合消息。但是请注意，为了防止Dos攻击服务器，需要合理的限制消息的大小。应设置多大取决于实际的需求，当然也得有足够的内存可用。

8.2.3 HTTP压缩

使用HTTP时建议压缩数据以减少传输流量，压缩数据会增加CPU负载，现在的硬件设施都很强大，大多数时候压缩数据时一个好主意。Netty支持“gzip”和“deflate”，为此提供了两个ChannelHandler实现分别用于压缩和解压。看下面代码：

```
[java]    

01.  @Override
02.  protected void initChannel(Channel ch) throws Exception {
03.      ChannelPipeline pipeline = ch.pipeline();
04.      if (client) {
05.          pipeline.addLast("codec", new HttpClientCodec());
06.          // 添加解压缩Handler
07.          pipeline.addLast("decompressor", new HttpContentDecompressor());
08.      } else {
09.          pipeline.addLast("codec", new HttpServerCodec());
10.          // 添加解压缩Handler
11.          pipeline.addLast("decompressor", new HttpContentDecompressor());
12.      }
13.      pipeline.addLast("aggregator", new HttpObjectAggregator(512 * 1024));
14.  }
```

8.2.4 使用HTTPS

网络中传输的重要数据需要加密来保护，使用Netty提供的SslHandler可以很容易实现，看下面代码：

```
[java]    

01.  /**
02.   * 使用SSL对HTTP消息加密
03.   *
04.   * @author c.k
05.   *
06.   */
07. public class HttpsCodecInitializer extends ChannelInitializer<Channel> {
08.
09.     private final SSLContext context;
10.     private final boolean client;
11.
12.     public HttpsCodecInitializer(SSLContext context, boolean client) {
13.         this.context = context;
14.         this.client = client;
15.     }
16.
17.     @Override
```

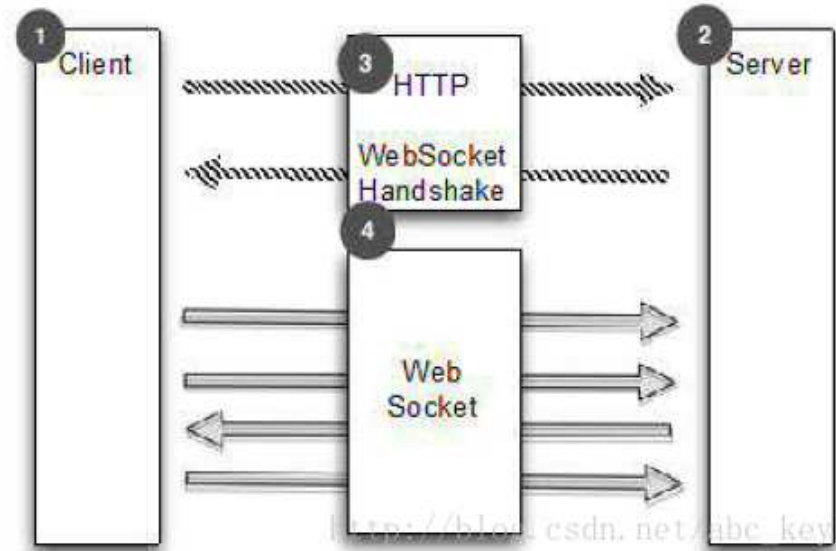


```
18.     protected void initChannel(Channel ch) throws Exception {
19.         SSLEngine engine = context.createSSLEngine();
20.         engine.setUseClientMode(client);
21.         ChannelPipeline pipeline = ch.pipeline();
22.         pipeline.addFirst("ssl", new SslHandler(engine));
23.         if (client) {
24.             pipeline.addLast("codec", new HttpClientCodec());
25.         } else {
26.             pipeline.addLast("codec", new HttpServerCodec());
27.         }
28.     }
29.
30. }
```

8.2.5 WebSocket

HTTP是不错的协议，但是如果需要实时发布信息怎么做？有个做法就是客户端一直轮询请求服务器，这种方式虽然可以达到目的，但是其缺点很多，也不是优秀的解决方案，为了解决这个问题，便出现了WebSocket。

WebSocket允许数据双向传输，而不需要请求-响应模式。早期的WebSocket只能发送文本数据，然后现在不仅可以发送文本数据，也可以发送二进制数据，这使得可以使用WebSocket构建你想要的程序。下图是WebSocket的通信示例图：



在应用程序中添加WebSocket支持很容易，Netty附带了WebSocket的支持，通过ChannelHandler来实现。使用WebSocket有不同的消息类型需要处理。下面列表列出了Netty中WebSocket类型：

- BinaryWebSocketFrame，包含二进制数据
- TextWebSocketFrame，包含文本数据
- ContinuationWebSocketFrame，包含二进制数据或文本数据，BinaryWebSocketFrame和TextWebSocketFrame的结合体
- CloseWebSocketFrame，WebSocketFrame代表一个关闭请求，包含关闭状态码和短语
- PingWebSocketFrame，WebSocketFrame要求PongWebSocketFrame发送数据
- PongWebSocketFrame，WebSocketFrame要求PingWebSocketFrame响应

为了简化，我们只看看如何使用WebSocket服务器。客户端使用可以看Netty自带的WebSocket例子。
Netty提供了许多方法来使用WebSocket，但最简单常用的方法是使用WebSocketServerProtocolHandler。看下面代码：

```
[java]
01.  /**
02.   * WebSocket Server，若想使用SSL加密，将SslHandler加载ChannelPipeline的最前面即可
03.   * @author c.k
04.   *
05.   */
06.  public class WebSocketServerInitializer extends ChannelInitializer<Channel> {
07.
08.      @Override
09.      protected void initChannel(Channel ch) throws Exception {
10.          ch.pipeline().addLast(new HttpServerCodec(),
11.                                new HttpObjectAggregator(65536),
12.                                new WebSocketServerProtocolHandler("/websocket"),
13.                                new TextFrameHandler(),
14.                                new BinaryFrameHandler(),
15.                                new ContinuationFrameHandler());
16.      }
17.
18.      public static final class TextFrameHandler extends SimpleChannelInboundHandler<TextWebSocketFrame> {
19.          @Override
20.          protected void channelRead0(ChannelHandlerContext ctx, TextWebSocketFrame msg) throws Exception {
21.              // handler text frame

```

```
22.     }
23. }
24.
25. public static final class BinaryFrameHandler extends SimpleChannelInboundHandler<BinaryWebSocketFrame>{
26.     @Override
27.     protected void channelRead0(ChannelHandlerContext ctx, BinaryWebSocketFrame msg) throws Exception {
28.         //handler binary frame
29.     }
30. }
31.
32. public static final class ContinuationFrameHandler extends SimpleChannelInboundHandler<ContinuationWebSocketFrame>{
33.     @Override
34.     protected void channelRead0(ChannelHandlerContext ctx, ContinuationWebSocketFrame msg) throws Exception {
35.         //handler continuation frame
36.     }
37. }
38. }
```

8.2.6 SPDY

SPDY（读作“SPeeDY”）是Google开发的基于TCP的应用层协议，用以最小化网络延迟，提升网络速度，优化用户的网络使用体验。SPDY并不是一种用于替代HTTP的协议，而是对HTTP协议的增强。新协议的功能包括数据流的多路复用、请求优先级以及HTTP报头压缩。谷歌表示，引入SPDY协议后，在实验室测试中页面加载速度比原先快64%。

SPDY的定位：

- 将页面加载时间减少50%。
- 最大限度地减少部署的复杂性。SPDY使用TCP作为传输层，因此无需改变现有的网络设施。
- 避免网站开发者改动内容。 支持SPDY唯一需要变化的是客户端代理和Web服务器应用程序。

SPDY实现技术：

- 单个TCP连接支持并发的HTTP请求。
- 压缩报头和去掉不必要的头部来减少当前HTTP使用的带宽。
- 定义一个容易实现，在服务器端高效率的协议。通过减少边缘情况、定义易解析的消息格式来减少HTTP的复杂性。
- 强制使用SSL，让SSL协议在现存的网络设施下有更好的安全性和兼容性。
- 允许服务器在需要时发起对客户端的连接并推送数据。

SPDY具体的细节知识及使用可以查阅相关资料，这里不作赘述了。

8.3 处理空闲连接和超时

处理空闲连接和超时是网络应用程序的核心部分。当发送一条消息后，可以检测连接是否还处于活跃状态，若很长时间没用了就可以断开连接。Netty提供了很好的解决方案，有三种不同的ChannelHandler处理闲置和超时连接：

- IdleStateHandler，当一个通道没有进行读写或运行了一段时间后出发IdleStateEvent
- ReadTimeoutHandler，在指定时间内没有接收到任何数据将抛出ReadTimeoutException
- WriteTimeoutHandler，在指定时间内有写入数据将抛出WriteTimeoutException

最常用的是IdleStateHandler，下面代码显示了如何使用IdleStateHandler，如果60秒内没有接收数据或发送数据，操作将失败，连接将关闭：

```
[ java ]
01. public class IdleStateHandlerInitializer extends ChannelInitializer<Channel> {
02.
03.     @Override
04.     protected void initChannel(Channel ch) throws Exception {
05.         ChannelPipeline pipeline = ch.pipeline();
06.         pipeline.addLast(new IdleStateHandler(0, 0, 60, TimeUnit.SECONDS));
07.         pipeline.addLast(new HeartbeatHandler());
08.     }
09.
10.     public static final class HeartbeatHandler extends ChannelInboundHandlerAdapter {
11.         private static final ByteBuf HEARTBEAT_SEQUENCE = Unpooled.unreleasableBuffer(Unpooled.copiedBuffer(
12.             "HEARTBEAT", CharsetUtil.UTF_8));
13.
14.         @Override
15.         public void userEventTriggered(ChannelHandlerContext ctx, Object evt) throws Exception {
16.             if (evt instanceof IdleStateEvent) {
17.                 ctx.writeAndFlush(HEARTBEAT_SEQUENCE.duplicate()).addListener(ChannelFutureListener.CLOSE_ON_FAILURE);
18.             } else {
19.                 super.userEventTriggered(ctx, evt);
20.             }
21.         }
22.     }
```

8.4 解码分隔符和基于长度的协议

使用Netty时会遇到需要解码以分隔符和长度为基础的协议，本节讲解Netty如何解码这些协议。

8.4.1 分隔符协议

经常需要处理分隔符协议或创建基于它们的协议，例如SMTP、POP3、IMAP、Telnet等等；Netty附带的handlers可以很容易的提取一些序列分隔：

- `DelimiterBasedFrameDecoder`，解码器，接收`ByteBuf`由一个或多个分隔符拆分，如NUL或换行符
- `LineBasedFrameDecoder`，解码器，接收`ByteBuf`以分割线结束，如“\n”和“\r\n”

下图显示了使用“\r\n”分隔符的处理：



下面代码显示使用`LineBasedFrameDecoder`提取“\r\n”分隔帧：

```
[java]
01. /**
02.  * 处理换行分隔符消息
03.  * @author c.k
04.  *
05.  */
06. public class LineBasedHandlerInitializer extends ChannelInitializer<Channel> {
07.
08.     @Override
09.     protected void initChannel(Channel ch) throws Exception {
10.         ch.pipeline().addLast(new LineBasedFrameDecoder(65 * 1204), new FrameHandler());
11.     }
12.
13.     public static final class FrameHandler extends SimpleChannelInboundHandler<ByteBuf> {
14.
15.         @Override
16.         protected void channelRead0(ChannelHandlerContext ctx, ByteBuf msg) throws Exception {
17.             // do something with the frame
18.         }
19.     }
20. }
```

如果框架的东西除了换行符还有别的分隔符，可以使用`DelimiterBasedFrameDecoder`，只需要将分隔符传递到构造方法中。如果想实现自己的以分隔符为基础的协议，这些解码器是有用的。例如，现在有个协议，它只处理命令，这些命令由名称和参数形成，名称和参数由一个空格分隔，实现这个需求的代码如下：

```
[java]
01. /**
02.  * 自定义以分隔符为基础的协议
03.  * @author c.k
04.  *
05.  */
06. public class CmdHandlerInitializer extends ChannelInitializer<Channel> {
07.
08.     @Override
09.     protected void initChannel(Channel ch) throws Exception {
10.         ch.pipeline().addLast(new CmdDecoder(65 * 1024), new CmdHandler());
11.     }
12.
13.     public static final class Cmd {
14.         private final ByteBuf name;
15.         private final ByteBuf args;
16.
17.         public Cmd(ByteBuf name, ByteBuf args) {
18.             this.name = name;
19.             this.args = args;
20.         }
21.
22.         public ByteBuf getName() {
23.             return name;
24.         }
25.     }
26. }
```

```
25.
26.     public ByteBuf getArgs() {
27.         return args;
28.     }
29. }
30.
31. public static final class CmdDecoder extends LineBasedFrameDecoder {
32.
33.     public CmdDecoder(int maxLength) {
34.         super(maxLength);
35.     }
36.
37.     @Override
38.     protected Object decode(ChannelHandlerContext ctx, ByteBuf buffer) throws Exception {
39.         ByteBuf frame = (ByteBuf) super.decode(ctx, buffer);
40.         if (frame == null) {
41.             return null;
42.         }
43.         int index = frame.indexOf(frame.readerIndex(), frame.writerIndex(), (byte) ' ');
44.         return new Cmd(frame.slice(frame.readerIndex(), index), frame.slice(index + 1, frame.writerIndex()));
45.     }
46. }
47.
48. public static final class CmdHandler extends SimpleChannelInboundHandler<Cmd> {
49.     @Override
50.     protected void channelRead0(ChannelHandlerContext ctx, Cmd msg) throws Exception {
51.         // do something with the command
52.     }
53. }
54.
55. }
```

8.4.2 长度为基础的协议

一般经常会碰到以长度为基础的协议，对于这种情况Netty有两个不同的解码器可以帮助我们来解码：

- FixedLengthFrameDecoder
- LengthFieldBasedFrameDecoder

下图显示了FixedLengthFrameDecoder的处理流程：

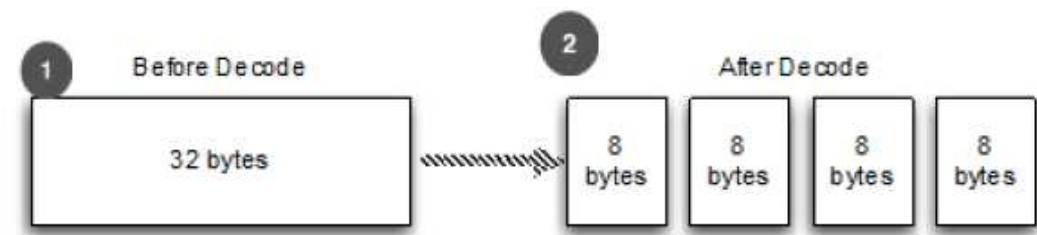


Figure 8.6 Message of fixed size of 8 bytes

如上图所示，FixedLengthFrameDecoder提取固定长度，例子中的是8字节。大部分时候帧的大小被编码在头部，这种情况可以使用LengthFieldBasedFrameDecoder，它会读取头部长度并提取帧的长度。下图显示了它是如何工作的：

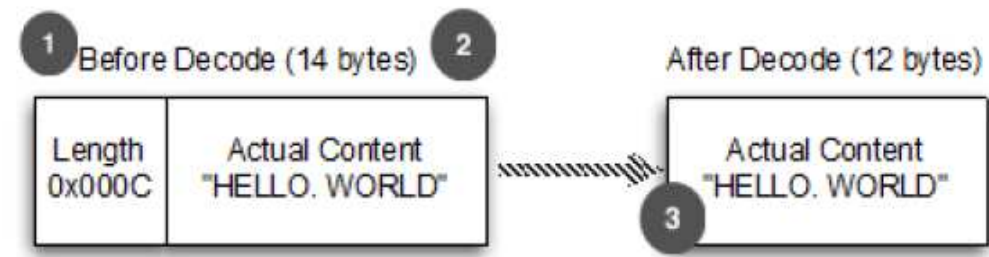


Figure 8.7 Message that has fixed size encoded in the header

如果长度字段是提取框架的一部分，可以在LengthFieldBasedFrameDecoder的构造方法中配置，还可以指定提供的长度。FixedLengthFrameDecoder很容易使用，我们重点讲解LengthFieldBasedFrameDecoder。下面代码显示如何使用LengthFieldBasedFrameDecoder提取8字节长度：

```
[ java ]
01. public class LengthBasedInitializer extends ChannelInitializer<Channel> {
02.
03.     @Override
04.     protected void initChannel(Channel ch) throws Exception {
05.         ch.pipeline().addLast(new LengthFieldBasedFrameDecoder(65*1024, 0, 8))
06.             .addLast(new FrameHandler());
07.     }
08. }
```

```
09.         public static final class FrameHandler extends SimpleChannelInboundHandler<ByteBuf>{
10.             @Override
11.             protected void channelRead0(ChannelHandlerContext ctx, ByteBuf msg) throws Exception {
12.                 //do something with the frame
13.             }
14.         }
15.     }
```

8.5 写大数据

写大量的数据的一个有效的方法是使用异步框架，如果内存和网络都处于饱满负荷状态，你需要停止写，否则会报OutOfMemoryError。Netty提供了写文件内容时zero-memory-copy机制，这种方法再将文件内容写到网络堆栈空间时可以获得最大的性能。使用零拷贝写文件的内容时通过DefaultFileRegion、ChannelHandlerContext、ChannelPipeline，看下面代码：

```
[java]
01. @Override
02. public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {
03.     File file = new File("test.txt");
04.     FileInputStream fis = new FileInputStream(file);
05.     FileRegion region = new DefaultFileRegion(fis.getChannel(), 0, file.length());
06.     Channel channel = ctx.channel();
07.     channel.writeAndFlush(region).addListener(new ChannelFutureListener() {
08.
09.         @Override
10.         public void operationComplete(ChannelFuture future) throws Exception {
11.             if(!future.isSuccess()){
12.                 Throwable cause = future.cause();
13.                 // do something
14.             }
15.         }
16.     });
17. }
```

如果只想发送文件中指定的数据块应该怎么做呢？Netty提供了ChunkedWriteHandler，允许通过处理ChunkedInput来写大的数据块。下面是ChunkedInput的一些实现类：

- ChunkedFile
- ChunkedNioFile
- ChunkedStream
- ChunkedNioStream

看下面代码：

```
[java]
01. public class ChunkedWriteHandlerInitializer extends ChannelInitializer<Channel> {
02.     private final File file;
03.
04.     public ChunkedWriteHandlerInitializer(File file) {
05.         this.file = file;
06.     }
07.
08.     @Override
09.     protected void initChannel(Channel ch) throws Exception {
10.         ch.pipeline().addLast(new ChunkedWriteHandler())
11.             .addLast(new WriteStreamHandler());
12.     }
13.
14.     public final class WriteStreamHandler extends ChannelInboundHandlerAdapter {
15.         @Override
16.         public void channelActive(ChannelHandlerContext ctx) throws Exception {
17.             super.channelActive(ctx);
18.             ctx.writeAndFlush(new ChunkedStream(new FileInputStream(file)));
19.         }
20.     }
21. }
```

8.6 序列化数据

开发网络程序过程中，很多时候需要传输结构化对象数据POJO,Java中提供了ObjectInputStream和ObjectOutputStream及其他的一些对象序列化接口。Netty中提供基于JDK序列化接口的序列化接口。

8.6.1 普通的JDK序列化

如果你使用ObjectInputStream和ObjectOutputStream，并且需要保持兼容性，不想有外部依赖，那么JDK的序列化是首选。Netty提供了

下面的一些接口，这些接口放在io.netty.handler.codec.serialization包下面：

- CompatibleObjectEncoder
- CompactObjectInputStream
- CompactObjectOutputStream
- ObjectEncoder
- ObjectDecoder
- ObjectEncoderOutputStream
- ObjectDecoderInputStream

8.6.2 通过JBoss编组序列化

如果你想使用外部依赖的接口，JBoss编组是个好方法。JBoss Marshalling序列化的速度是JDK的3倍，并且序列化的结构更紧凑，从而使序列化后的数据更小。Netty附带了JBoss编组序列化的实现，这些实现接口放在io.netty.handler.codec.marshalling包下面：

- CompatibleMarshallingEncoder
- CompatibleMarshallingDecoder
- MarshallingEncoder
- MarshallingDecoder

看下面代码：

```
[java]    

01.  /**
02.   * 使用JBoss Marshalling
03.   * @author c.k
04.   *
05.   */
06. public class MarshallingInitializer extends ChannelInitializer<Channel> {
07.     private final MarshallerProvider marshallerProvider;
08.     private final UnmarshallerProvider unmarshallerProvider;
09.
10.     public MarshallingInitializer(MarshallerProvider marshallerProvider, UnmarshallerProvider unmarshallerProvider) {
11.         this.marshallerProvider = marshallerProvider;
12.         this.unmarshallerProvider = unmarshallerProvider;
13.     }
14.
15.     @Override
16.     protected void initChannel(Channel ch) throws Exception {
17.         ch.pipeline().addLast(new MarshallingDecoder(unmarshallerProvider))
18.             .addLast(new MarshallingEncoder(marshallerProvider))
19.             .addLast(new ObjectHandler());
20.     }
21.
22.     public final class ObjectHandler extends SimpleChannelInboundHandler<Serializable> {
23.         @Override
24.         protected void channelRead0(ChannelHandlerContext ctx, Serializable msg) throws Exception {
25.             // do something
26.         }
27.     }
28. }
```

8.6.3 使用ProtoBuf序列化

最有一个序列化方案是Netty自带的ProtoBuf。protobuf是Google开源的一种编码和解码技术，它的作用是使序列化数据更高效。并且谷歌提供了protobuf的不同语言的实现，所以protobuf在跨平台项目中是非常好的选择。Netty自带的protobuf放在io.netty.handler.codec.protobuf包下面：

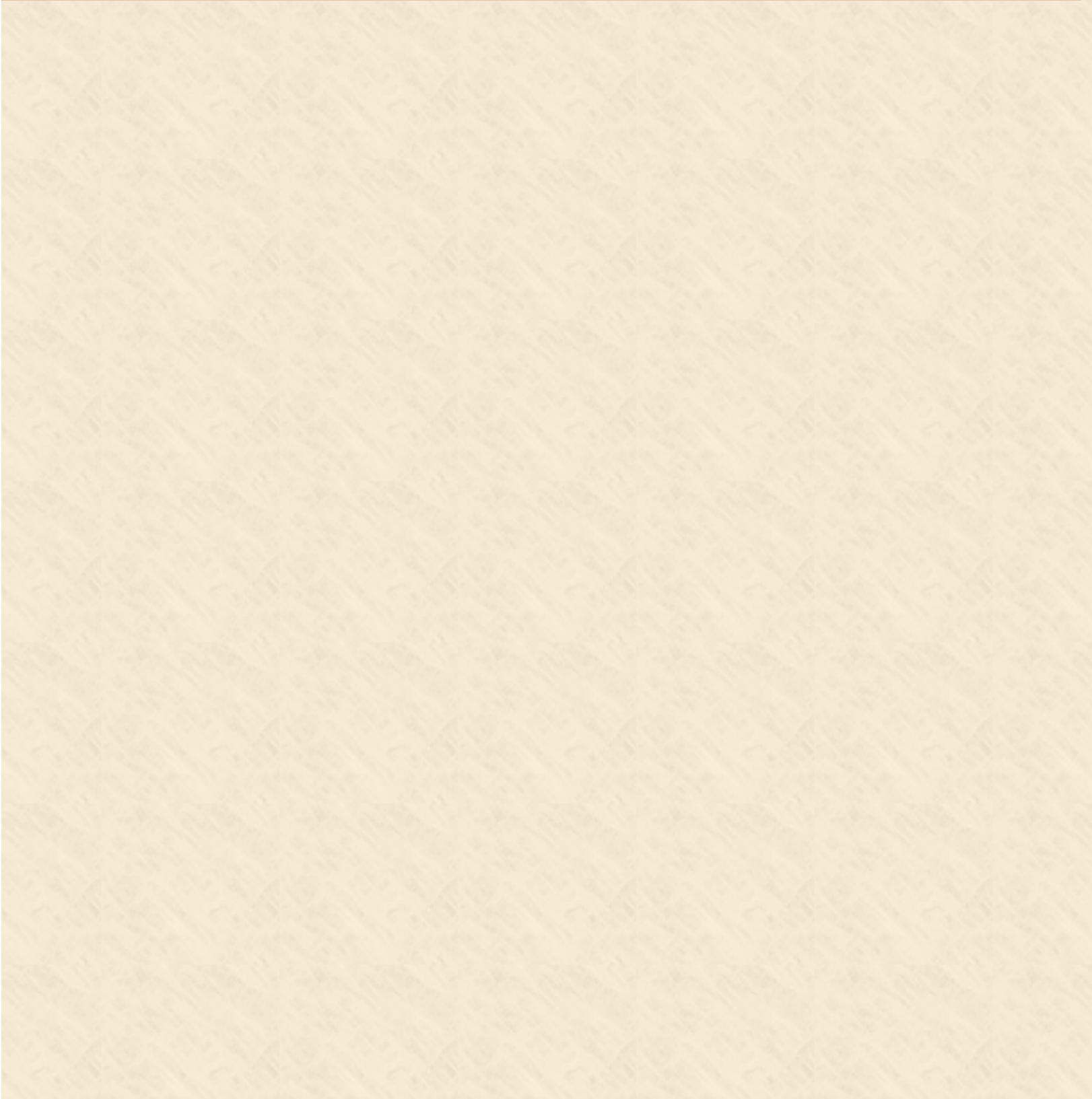
- ProtobufDecoder
- ProtobufEncoder
- ProtobufVarint32FrameDecoder
- ProtobufVarint32LengthFieldPrepender

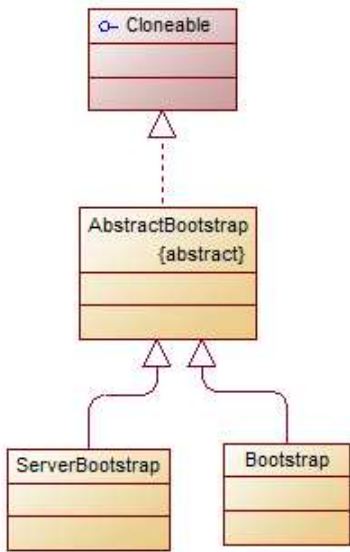
看下面代码：

```
[java]    

01.  /**
02.   * 使用protobuf序列化数据，进行编码解码
03.   * 注意：使用protobuf需要protobuf-java-2.5.0.jar
```

```
04.  * @author Administrator
05.  *
06.  */
07.  public class ProtoBufInitializer extends ChannelInitializer<Channel> {
08.
09.      private final MessageLite lite;
10.
11.      public ProtoBufInitializer(MessageLite lite) {
12.          this.lite = lite;
13.      }
14.
15.      @Override
16.      protected void initChannel(Channel ch) throws Exception {
17.          ch.pipeline().addLast(new ProtobufVarint32FrameDecoder())
18.              .addLast(new ProtobufEncoder())
19.              .addLast(new ProtobufDecoder(lite))
20.              .addLast(new ObjectHandler());
21.      }
22.
23.      public final class ObjectHandler extends SimpleChannelInboundHandler<Serializable> {
24.          @Override
25.          protected void channelRead0(ChannelHandlerContext ctx, Serializable msg) throws Exception {
26.              // do something
27.          }
28.      }
29.  }
```





我们在前面讨论了许多用于客户端和服务器的知识，为了对客户端和服务端之间的关系提供了一个共同点，Netty使用AbstractBootstrap类。通过一个共同的父类，在本章中讨论的客户端和服务器的引导程序能够重复使用通用功能，而无需复制代码或逻辑。通常情况下，多个通道使用相同或非常类似的设置时有必要的。而不是为每一个通道创建一个新的引导，Netty使得AbstractBootstrap可复制。也就是说克隆一个已配置的引导，其返回的是一个可重用而无需配置的引导。Netty的克隆操作只能浅拷贝引导的EventLoopGroup，也就是说EventLoopGroup在所有的克隆的通道中是共享的。这是一个好事情，克隆的通道一般是短暂的，例如一个通道创建一个HTTP请求。

本章主要讲解Bootstrap和ServerBootstrap，首先我们来看看ServerBootstrap。

9.2 引导客户端和无连接协议

当需要引导客户端或一些无连接协议时，需要使用Bootstrap类。

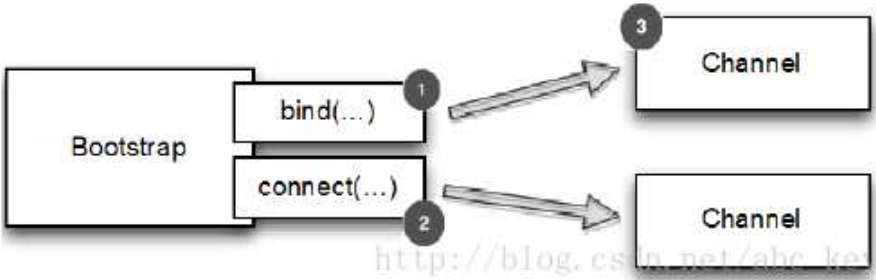
9.2.1 引导客户端的方法

创建Bootstrap实例使用new关键字，下面是Bootstrap的方法：

- group(...), 设置EventLoopGroup,EventLoopGroup用来处理所有通道的IO事件
- channel(...), 设置通道类型
- channelFactory(...), 使用ChannelFactory来设置通道类型
- localAddress(...), 设置本地地址，也可以通过bind(...)或connect(...)
- option(ChannelOption<T>, T), 设置通道选项，若使用null，则删除上一个设置的ChannelOption
- attr(AttributeKey<T>, T), 设置属性到Channel，若值为null，则指定键的属性被删除
- handler(ChannelHandler), 设置ChannelHandler用于处理请求事件
- clone(), 深度复制Bootstrap，Bootstrap的配置相同
- remoteAddress(...), 设置连接地址
- connect(...), 连接远程通道
- bind(...), 创建一个新的Channel并绑定

9.2.2 怎么引导客户端

引导负责客户端通道连接或断开连接，因此它将在调用bind(...)或connect(...)后创建通道。下图显示了如何工作：



下面代码显示了引导客户端使用NIO TCP传输：

```
[html]
01. package netty.in.action;
02.
03. import io.netty.bootstrap.Bootstrap;
04. import io.netty.buffer.ByteBuf;
05. import io.netty.channel.ChannelFuture;
06. import io.netty.channel.ChannelFutureListener;
07. import io.netty.channel.ChannelHandlerContext;
08. import io.netty.channel.EventLoopGroup;
09. import io.netty.channel.SimpleChannelInboundHandler;
10. import io.netty.channel.nio.NioEventLoopGroup;
11. import io.netty.channel.socket.nio.NioSocketChannel;
```

```
12.
13.  /**
14.   * 引导配置客户端
15.   *
16.   * @author c.k
17.   *
18.   */
19. public class BootstrappingClient {
20.
21.     public static void main(String[] args) throws Exception {
22.         EventLoopGroup group = new NioEventLoopGroup();
23.         Bootstrap b = new Bootstrap();
24.         b.group(group).channel(NioSocketChannel.class).handler(new SimpleChannelInboundHandler<ByteBuf>() {
25.             @Override
26.             protected void channelRead0(ChannelHandlerContext ctx, ByteBuf msg) throws Exception {
27.                 System.out.println("Received data");
28.                 msg.clear();
29.             }
30.         });
31.         ChannelFuture f = b.connect("127.0.0.1", 2048);
32.         f.addListener(new ChannelFutureListener() {
33.             @Override
34.             public void operationComplete(ChannelFuture future) throws Exception {
35.                 if (future.isSuccess()) {
36.                     System.out.println("connection finished");
37.                 } else {
38.                     System.out.println("connection failed");
39.                     future.cause().printStackTrace();
40.                 }
41.             }
42.         });
43.     }
44. }
```

9.2.3 选择兼容通道实现

Channel的实现和EventLoop的处理过程在EventLoopGroup中必须兼容，哪些Channel是和EventLoopGroup是兼容的可以查看API文档。经验显示，相兼容的实现一般在同一个包下面，例如使用NioEventLoop，NioEventLoopGroup和NioServerSocketChannel在一起。请注意，这些都是前缀“Nio”，然后不会用这些代替另一个实现和另一个前缀，如“Oio”，也就是说OioEventLoopGroup和NioServerSocketChannel是不相容的。

Channel和EventLoopGroup的EventLoop必须相容，例如NioEventLoop、NioEventLoopGroup、NioServerSocketChannel是相容的，但是OioEventLoopGroup和NioServerSocketChannel是不相容的。从类名可以看出前缀是“Nio”的只能和“Nio”的一起使用，“Oio”前缀的只能和Oio*一起使用，将不相容的一起使用会导致错误异常，如OioSocketChannel和NioEventLoopGroup一起使用时会抛出异常：Exception in thread "main" java.lang.IllegalStateException: incompatible event loop type。

9.3 使用ServerBootstrap引导服务器

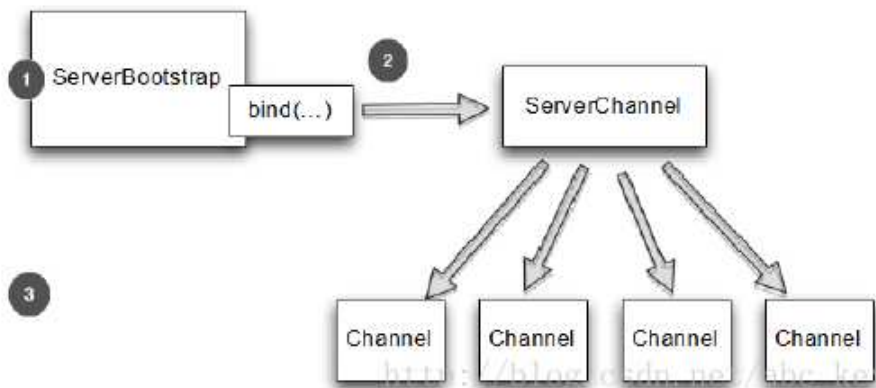
9.3.1 引导服务器的方法

先看看ServerBootstrap提供了哪些方法

- group(...)，设置EventLoopGroup事件循环组
- channel(...)，设置通道类型
- channelFactory(...)，使用ChannelFactory来设置通道类型
- localAddress(...)，设置本地地址，也可以通过bind(...)或connect(...)
- option(ChannelOption<T>, T)，设置通道选项，若使用null，则删除上一个设置的ChannelOption
- childOption(ChannelOption<T>, T)，设置子通道选项
- attr(AttributeKey<T>, T)，设置属性到Channel，若值为null，则指定键的属性被删除
- childAttr(AttributeKey<T>, T)，设置子通道属性
- handler(ChannelHandler)，设置ChannelHandler用于处理请求事件
- childHandler(ChannelHandler)，设置子ChannelHandler
- clone()，深度复制ServerBootstrap，且配置相同
- bind(...)，创建一个新的Channel并绑定

9.3.2 怎么引导服务器

下图显示ServerBootstrap管理子通道：



child*方法是在子Channel上操作，通过ServerChannel来管理。

下面代码显示使用ServerBootstrap引导配置服务器：

```
[java]
01. package netty.in.action;
02.
03. import io.netty.bootstrap.ServerBootstrap;
04. import io.netty.buffer.ByteBuf;
05. import io.netty.channel.ChannelFuture;
06. import io.netty.channel.ChannelFutureListener;
07. import io.netty.channel.ChannelHandlerContext;
08. import io.netty.channel.EventLoopGroup;
09. import io.netty.channel.SimpleChannelInboundHandler;
10. import io.netty.channel.nio.NioEventLoopGroup;
11. import io.netty.channel.socket.nio.NioServerSocketChannel;
12.
13. /**
14.  * 引导服务器配置
15.  * @author c.k
16.  *
17.  */
18. public class BootstrappingServer {
19.
20.     public static void main(String[] args) throws Exception {
21.         EventLoopGroup bossGroup = new NioEventLoopGroup(1);
22.         EventLoopGroup workerGroup = new NioEventLoopGroup();
23.         ServerBootstrap b = new ServerBootstrap();
24.         b.group(bossGroup, workerGroup).channel(NioServerSocketChannel.class)
25.           .childHandler(new SimpleChannelInboundHandler<ByteBuf>() {
26.               @Override
27.               protected void channelRead0(ChannelHandlerContext ctx, ByteBuf msg) throws Exception {
28.                   System.out.println("Received data");
29.                   msg.clear();
30.               }
31.           });
32.         ChannelFuture f = b.bind(2048);
33.         f.addListener(new ChannelFutureListener() {
34.             @Override
35.             public void operationComplete(ChannelFuture future) throws Exception {
36.                 if (future.isSuccess()) {
37.                     System.out.println("Server bound");
38.                 } else {
39.                     System.err.println("bound fail");
40.                     future.cause().printStackTrace();
41.                 }
42.             }
43.         });
44.     }
45. }
```

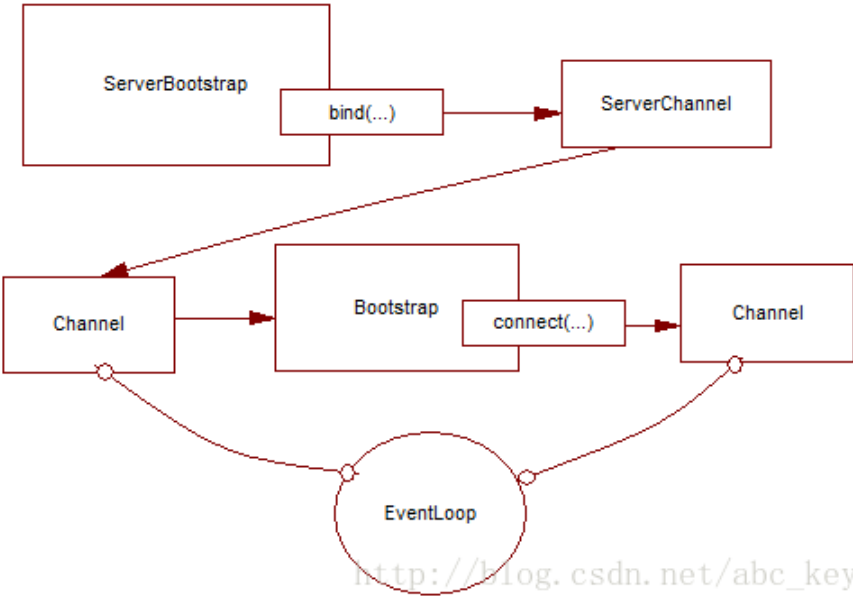
9.4 从Channel引导客户端

有时候需要从另一个Channel引导客户端，例如写一个代理或需要从其他系统检索数据。从其他系统获取数据时比较常见的，有很多Netty应用程序必须要和企业现有的系统集成，如Netty程序与内部系统进行身份验证，查询数据库等。

当然，你可以创建一个新的引导，这样做没有什么不妥，只是效率不高，因为要为新创建的客户端通道使用另一个EventLoop，如果需要在已接受的通道和客户端通道之间交换数据则需要切换上下文线程。Netty在这方面进行了优化，可以讲已接受的通道通过eventLoop(...)传递到EventLoop，从而使客户端通道在相同的EventLoop里运行。这消除了额外的上下文切换工作，因为EventLoop继承于EventLoopGroup。除了消除上下文切换，还可以在不需要创建多个线程的情况下使用引导。

为什么要共享EventLoop呢？一个EventLoop由一个线程执行，共享EventLoop可以确定所有的Channel都分配给同一线程的EventLoop，这样就避免了不同线程之间切换上下文，从而减少资源开销。

下图显示相同的EventLoop管理两个Channel:



看下面代码：

```
[java]
01. package netty.in.action;
02.
03. import java.net.InetSocketAddress;
04.
05. import io.netty.bootstrap.Bootstrap;
06. import io.netty.bootstrap.ServerBootstrap;
07. import io.netty.buffer.ByteBuf;
08. import io.netty.channel.ChannelFuture;
09. import io.netty.channel.ChannelFutureListener;
10. import io.netty.channel.ChannelHandlerContext;
11. import io.netty.channel.EventLoopGroup;
12. import io.netty.channel.SimpleChannelInboundHandler;
13. import io.netty.channel.nio.NioEventLoopGroup;
14. import io.netty.channel.socket.nio.NioServerSocketChannel;
15. import io.netty.channel.socket.nio.NioSocketChannel;
16.
17. /**
18.  * 从Channel 引导客户端
19.  *
20.  * @author c.k
21.  *
22.  */
23. public class BootstrappingFromChannel {
24.
25.     public static void main(String[] args) throws Exception {
26.         EventLoopGroup bossGroup = new NioEventLoopGroup(1);
27.         EventLoopGroup workerGroup = new NioEventLoopGroup();
28.         ServerBootstrap b = new ServerBootstrap();
29.         b.group(bossGroup, workerGroup).channel(NioServerSocketChannel.class)
30.         .childHandler(new SimpleChannelInboundHandler<ByteBuf>() {
31.             ChannelFuture connectFuture;
32.
33.             @Override
34.             public void channelActive(ChannelHandlerContext ctx) throws Exception {
35.                 Bootstrap b = new Bootstrap();
36.                 b.channel(NioSocketChannel.class).handler(
37.                     new SimpleChannelInboundHandler<ByteBuf>() {
38.                         @Override
39.                         protected void channelRead0(ChannelHandlerContext ctx,
40.                             ByteBuf msg) throws Exception {
41.                             System.out.println("Received data");
42.                             msg.clear();
43.                         }
44.                     });
45.                 b.group(ctx.channel().eventLoop());
46.                 connectFuture = b.connect(new InetSocketAddress("127.0.0.1", 2048));
47.             }
48.
49.             @Override
50.             protected void channelRead0(ChannelHandlerContext ctx, ByteBuf msg)
51.                 throws Exception {
52.                 if (connectFuture.isDone()) {
53.                     // do something with the data
54.                 }
55.             }
56.         }
57.     }
58. }
```

```
56.         });
57.         ChannelFuture f = b.bind(2048);
58.         f.addListener(new ChannelFutureListener() {
59.             @Override
60.             public void operationComplete(ChannelFuture future) throws Exception {
61.                 if (future.isSuccess()) {
62.                     System.out.println("Server bound");
63.                 } else {
64.                     System.err.println("bound fail");
65.                     future.cause().printStackTrace();
66.                 }
67.             }
68.         });
69.     }
70. }
```

9.5 添加多个ChannelHandler

在所有的例子代码中，我们在引导过程中通过handler(...)或childHandler(...)都只添加了一个ChannelHandler实例，对于简单的程序可能足够，但是对于复杂的程序则无法满足需求。例如，某个程序必须支持多个协议，如HTTP、WebSocket。若在一个ChannelHandler中处理这些协议将导致一个庞大而复杂的ChannelHandler。Netty通过添加多个ChannelHandler，从而使每个ChannelHandler分工明确，结构清晰。

Netty的一个优势是可以在ChannelPipeline中堆叠很多ChannelHandler并且可以最大程度的重用代码。如何添加多个ChannelHandler呢？Netty提供ChannelInitializer抽象类用来初始化ChannelPipeline中的ChannelHandler。ChannelInitializer是一个特殊的ChannelHandler，通道被注册到EventLoop后就会调用ChannelInitializer，并允许将ChannelHandler添加到ChannelPipeline；完成初始化通道后，这个特殊的ChannelHandler初始化器会从ChannelPipeline中自动删除。

听起来很复杂，其实很简单，看下面代码：

```
[java]

01. package netty.in.action;
02.
03. import io.netty.bootstrap.ServerBootstrap;
04. import io.netty.channel.Channel;
05. import io.netty.channel.ChannelFuture;
06. import io.netty.channel.ChannelInitializer;
07. import io.netty.channel.EventLoopGroup;
08. import io.netty.channel.nio.NioEventLoopGroup;
09. import io.netty.channel.socket.nio.NioServerSocketChannel;
10. import io.netty.handler.codec.http.HttpClientCodec;
11. import io.netty.handler.codec.http.HttpObjectAggregator;
12.
13. /**
14.  * 使用ChannelInitializer初始化ChannelHandler
15.  * @author c.k
16.  *
17.  */
18. public class InitChannelExample {
19.
20.     public static void main(String[] args) throws Exception {
21.         EventLoopGroup bossGroup = new NioEventLoopGroup(1);
22.         EventLoopGroup workerGroup = new NioEventLoopGroup();
23.         ServerBootstrap b = new ServerBootstrap();
24.         b.group(bossGroup, workerGroup).channel(NioServerSocketChannel.class)
25.           .childHandler(new ChannelInitializerImpl());
26.         ChannelFuture f = b.bind(2048).sync();
27.         f.channel().closeFuture().sync();
28.     }
29.
30.     static final class ChannelInitializerImpl extends ChannelInitializer<Channel>{
31.         @Override
32.         protected void initChannel(Channel ch) throws Exception {
33.             ch.pipeline().addLast(new HttpClientCodec())
34.               .addLast(new HttpObjectAggregator(Integer.MAX_VALUE));
35.         }
36.     }
37.
38. }
```

9.6 使用通道选项和属性

比较麻烦的是创建通道后不得不手动配置每个通道，为了避免这种情况，Netty提供了ChannelOption来帮助引导配置。这些选项会自动应用到引导创建的所有通道，可用的各种选项可以配置底层连接的详细信息，如通道“keep-alive(保持活跃)”或“timeout(超时)”的特性。

Netty应用程序通常会与组织或公司其他的软件进行集成，在某些情况下，Netty的组件如通道、传递和Netty正常生命周期外使用；在这样的情况下并不是所有的一般属性和数据时可用的。这只是一个例子，但在这样的情况下，Netty提供了通道属性(channel attributes)。

属性可以将数据和通道以一个安全的方式关联，这些属性只是作用于客户端和服务器的通道。例如，例如客户端请求web服务器应用程序，为了跟踪通道属于哪个用户，应用程序可以存储用的ID作为通道的一个属性。任何对象或数据都可以使用属性被关联到一个通道。

使用ChannelOption和属性可以让事情变得很简单，例如Netty WebSocket服务器根据用户自动路由消息，通过使用属性，应用程序能在通道存储用户ID以确定消息应该发送到哪里。应用程序可以通过使用一个通道选项进一步自动化，给定时间内没有收到消息将自动断开连接。看下面代码：

```
[ java ]
01. public static void main(String[] args) {
02.     // 创建属性键对象
03.     final AttributeKey<Integer> id = AttributeKey.valueOf("ID");
04.     // 客户端引导对象
05.     Bootstrap b = new Bootstrap();
06.     // 设置Event Loop，设置通道类型
07.     b.group(new NioEventLoopGroup()).channel(NioSocketChannel.class)
08.     // 设置Channel Handler
09.     .handler(new SimpleChannelInboundHandler<ByteBuf>() {
10.         @Override
11.         protected void channelRead0(ChannelHandlerContext ctx, ByteBuf msg)
12.             throws Exception {
13.             System.out.println("Reveived data");
14.             msg.clear();
15.         }
16.
17.         @Override
18.         public void channelRegistered(ChannelHandlerContext ctx) throws Exception {
19.             // 通道注册后执行，获取属性值
20.             Integer idValue = ctx.channel().attr(id).get();
21.             System.out.println(idValue);
22.             // do something with the idValue
23.         }
24.     });
25.     // 设置通道选项，在通道注册后或被创建后设置
26.     b.option(ChannelOption.SO_KEEPALIVE, true).option(ChannelOption.CONNECT_TIMEOUT_MILLIS, 5000);
27.     // 设置通道属性
28.     b.attr(id, 123456);
29.     ChannelFuture f = b.connect("www.manning.com", 80);
30.     f.syncUninterruptibly();
31. }
```

前面都是引导基于TCP的SocketChannel，引导也可以用于无连接的传输协议如UDP，Netty提供了DatagramChannel，唯一的区别是不会connecte(...)，只能bind(...)。看下面代码：

```
[ java ]
01. public static void main(String[] args) {
02.     Bootstrap b = new Bootstrap();
03.     b.group(new OioEventLoopGroup()).channel(OioDatagramChannel.class)
04.     .handler(new SimpleChannelInboundHandler<DatagramPacket>() {
05.         @Override
06.         protected void channelRead0(ChannelHandlerContext ctx, DatagramPacket msg)
07.             throws Exception {
08.             // do something with the packet
09.         }
10.     });
11.     ChannelFuture f = b.bind(new InetSocketAddress(0));
12.     f.addListener(new ChannelFutureListener() {
13.         @Override
14.         public void operationComplete(ChannelFuture future) throws Exception {
15.             if (future.isSuccess()) {
16.                 System.out.println("Channel bound");
17.             } else {
18.                 System.err.println("Bound attempt failed");
19.                 future.cause().printStackTrace();
20.             }
21.         }
22.     });
23. }
```

Netty有默认的配置设置，多数情况下，我们不需要改变这些配置，但是在需要时，我们可以细粒度的控制如何工作及处理数据。

9.7 Summary

In this chapter you learned how to bootstrap your Netty-based server and client implementation. You learned how you can specify configuration options that affect the and how you can use attributes to attach information to a channel and use it later. You also learned how to bootstrap connectionless protocol-based applications and how they are different from connection-based ones. The next chapters will focus on Netty in Action by using it to implement real-world applications. This will help you extract all interesting pieces for reuse in your next application. At this point you should be able to start coding!

目录	[-]
1. General	
2. 测试ChannelHandler	
1. 测试处理入站消息的handler	
2. 测试处理出站消息的handler	
3. 测试异常处理	
4. Summary	

本章介绍

- 单元测试
- EmbeddedChannel

学会了使用一个或多个ChannelHandler处理接收/发送数据消息，但是如何测试它们呢？Netty提供了2个额外的类使得测试ChannelHandler变得很容易，本章讲解如何测试Netty程序。测试使用JUnit4，如果不会用可以慢慢了解。JUnit4很简单，但是功能很强大。本章将重点讲解测试已实现的ChannelHandler和编解码器。

10.1 General

正如前面所学的，Netty提供了一个简单的方法在ChannelPipeline上“堆叠”不同的ChannelHandler实现。所有的ChannelHandler都会参与处理事件，这个设计允许独立出可重用的小逻辑块，它只处理一个任务。这不仅使代码更清晰，也更容易测试。

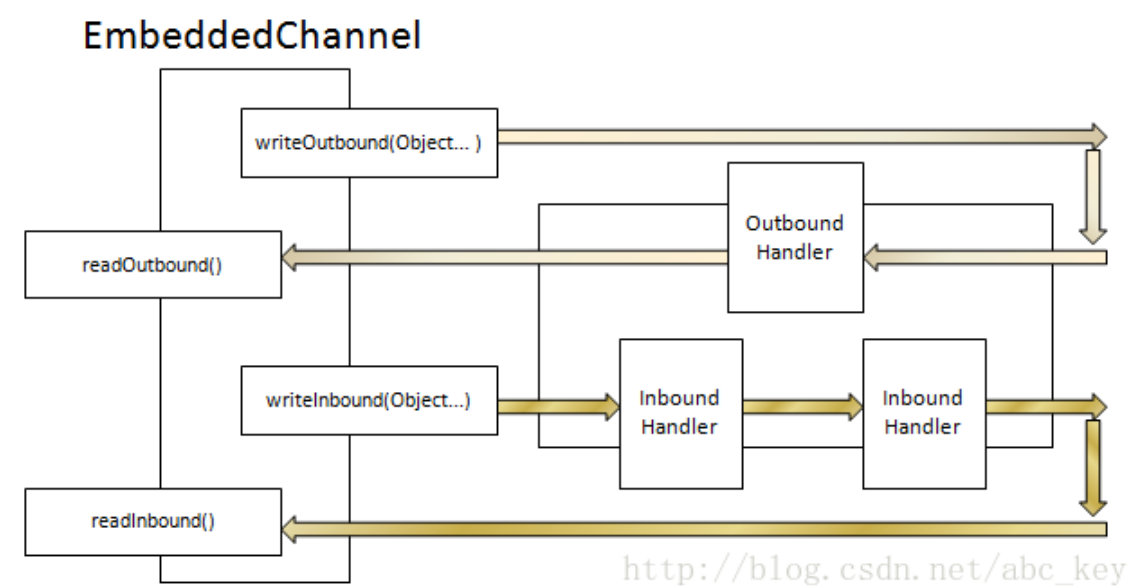
测试ChannelHandler可以通过使用“嵌入式”传输很容易的传递事件槽管道以测试你的实现。对于这个嵌入式传输，Netty提供了一个特定的Channel实现：EmbeddedChannel。但是它是如何工作的呢？EmbeddedChannel的工作非常简单，它允许写入入站或出站数据，然后检查ChannelPipeline的结束。这允许你检查消息编码/解码或触发ChannelHandler任何行为。

编写入站和出站的却别是什么？入站数据是通过ChannelInboundHandler处理，代表从远程对等通道读取数据；出站数据是通过ChannelOutboundHandler处理，代表写入数据到远程对等通道。因此测试ChannelHandler就会选择writeInbound(...)或writeOutbound()(或者都选择)。

EmbeddedChannel提供了下面一些方法：

- writeInbound(Object...)，写一个消息到入站通道
- writeOutbound(Object...)，写消息到出站通道
- readInbound()，从EmbeddedChannel读取入站消息，可能返回null
- readOutbound()，从EmbeddedChannel读取出站消息，可能返回null
- finish()，标示EmbeddedChannel已结束，任何写数据都会失败

为了更清楚的了解其处理过程，看下图：



如上图所示，使用writeOutbound(...)写消息到通道，消息在出站方法通过ChannelPipeline，之后就可以使用readOutbound()读取消息。着同样使用与入站，使用writeInbound(...)和readInbound()。处理入站和出站是相似的，它总是遍历整个ChannelPipeline直到ChannelPipeline结束，并将处理过的消息存储在EmbeddedChannel中。下面来看看如何测试你的逻辑。

10.2 测试ChannelHandler

测试ChannelHandler最好的选择是使用EmbeddedChannel。

10.2.1 测试处理入站消息的handler

我们来编写一个简单的ByteToMessageDecoder实现，有足够的可以读取时将产生固定大小的包，如果没有足够的可以读

取，则会等待下一个数据块并再次检查是否可以产生一个完整包。下图显示了重新组装接收的字节：



如上图所示，它可能会占用一个以上的“event”以获取足够的字节产生一个数据包，并将它传递到ChannelPipeline中的下一个ChannelHandler，看下面代码：

```
[java]
01. package netty.in.action;
02.
03. import java.util.List;
04.
05. import io.netty.buffer.ByteBuf;
06. import io.netty.channel.ChannelHandlerContext;
07. import io.netty.handler.codec.ByteToMessageDecoder;
08.
09. public class FixedLengthFrameDecoder extends ByteToMessageDecoder {
10.
11.     private final int frameLength;
12.
13.     public FixedLengthFrameDecoder(int frameLength) {
14.         if (frameLength <= 0) {
15.             throw new IllegalArgumentException(
16.                 "frameLength must be a positive integer: " + frameLength);
17.         }
18.         this.frameLength = frameLength;
19.     }
20.
21.     @Override
22.     protected void decode(ChannelHandlerContext ctx, ByteBuf in,
23.         List<Object> out) throws Exception {
24.         while (in.readableBytes() >= frameLength) {
25.             ByteBuf buf = in.readBytes(frameLength);
26.             out.add(buf);
27.         }
28.     }
29.
30. }
```

解码器的实现完成了，写一个单元测试的方法是个好主意。即使代码看起来没啥问题，但是也应该进行单元测试，这样能在部署到生产之前就发现问题。现在让我们来看看如何使用EmbeddedChannel来完成测试，看下面代码：

```
[java]
01. package netty.in.action;
02.
03. import io.netty.buffer.ByteBuf;
04. import io.netty.buffer.Unpooled;
05. import io.netty.channel.embedded.EmbeddedChannel;
06.
07. import org.junit.Assert;
08. import org.junit.Test;
09.
10. public class FixedLengthFrameDecoderTest {
11.
12.     @Test
13.     public void testFramesDecoded() {
14.         ByteBuf buf = Unpooled.buffer();
15.         for (int i = 0; i < 9; i++) {
16.             buf.writeByte(i);
17.         }
18.         ByteBuf input = buf.duplicate();
19.         EmbeddedChannel channel = new EmbeddedChannel(
20.             new FixedLengthFrameDecoder(3));
21.         // write bytes
22.         Assert.assertTrue(channel.writeInbound(input));
23.         Assert.assertTrue(channel.finish());
24.         // read message
25.         Assert.assertEquals(buf.readBytes(3), channel.readInbound());
26.         Assert.assertEquals(buf.readBytes(3), channel.readInbound());
27.         Assert.assertEquals(buf.readBytes(3), channel.readInbound());
28.         Assert.assertNull(channel.readInbound());
29.     }
30. }
```

```
31. @Test
32. public void testFramesDecoded2() {
33.     ByteBuf buf = Unpooled.buffer();
34.     for (int i = 0; i < 9; i++) {
35.         buf.writeByte(i);
36.     }
37.     ByteBuf input = buf.duplicate();
38.     EmbeddedChannel channel = new EmbeddedChannel(
39.         new FixedLengthFrameDecoder(3));
40.     Assert.assertFalse(channel.writeInbound(input.readBytes(2)));
41.     Assert.assertTrue(channel.writeInbound(input.readBytes(7)));
42.     Assert.assertTrue(channel.finish());
43.     Assert.assertEquals(buf.readBytes(3), channel.readInbound());
44.     Assert.assertEquals(buf.readBytes(3), channel.readInbound());
45.     Assert.assertEquals(buf.readBytes(3), channel.readInbound());
46.     Assert.assertNull(channel.readInbound());
47. }
48.
49. }
```

如上面代码，testFramesDecoded()方法想测试一个ByteBuf，这个ByteBuf包含9个可读字节，被解码成包含了3个可读字节的ByteBuf。你可能注意到，它写入9字节到通道是通过调用writeInbound()方法，之后再执行finish()来将EmbeddedChannel标记为已完成，最后调用readInbound()方法来获取EmbeddedChannel中的数据，直到没有可读字节。testFramesDecoded2()方法采取同样的方式，但有一个区别就是入站ByteBuf分两步写的，当调用writeInbound(input.readBytes(2))后返回false时，FixedLengthFrameDecoder值会产生输出，至少有3个字节是可读，testFramesDecoded2()测试的工作相当于testFramesDecoded()。

10.2.2 测试处理出站消息的handler

测试处理出站消息和测试处理入站消息不太一样，例如有一个继承MessageToMessageEncoder的AbsIntegerEncoder类，它所做的事情如下：

- 将已接收的数据flush()后将从ByteBuf读取所有整数并调用Math.abs(...)
- 完成后将字节写入ChannelPipeline中下一个ChannelHandler的ByteBuf中

看下图处理过程：



看下面代码：

```
[ java ] [ icon ] [ icon ] [ icon ] [ icon ]

01. package netty.in.action;
02.
03. import java.util.List;
04.
05. import io.netty.buffer.ByteBuf;
06. import io.netty.channel.ChannelHandlerContext;
07. import io.netty.handler.codec.MessageToMessageEncoder;
08.
09. public class AbsIntegerEncoder extends MessageToMessageEncoder<ByteBuf> {
10.     @Override
11.     protected void encode(ChannelHandlerContext ctx, ByteBuf msg,
12.         List<Object> out) throws Exception {
13.         while(msg.readableBytes() >= 4) {
14.             int value = Math.abs(msg.readInt());
15.             out.add(value);
16.         }
17.     }
18. }
```

下面代码是测试AbsIntegerEncoder：

```
[ java ] [ icon ] [ icon ] [ icon ] [ icon ]

01. package netty.in.action;
02.
03. import io.netty.buffer.ByteBuf;
04. import io.netty.buffer.Unpooled;
05. import io.netty.channel.embedded.EmbeddedChannel;
06.
07. import org.junit.Assert;
08. import org.junit.Test;
```

```
09.
10. public class AbsIntegerEncoderTest {
11.
12.     @Test
13.     public void testEncoded() {
14.         // 创建一个能容纳10个int的ByteBuffer
15.         ByteBuffer buf = Unpooled.buffer();
16.         for (int i = 1; i < 10; i++) {
17.             buf.writeInt(i * -1);
18.         }
19.         // 创建EmbeddedChannel对象
20.         EmbeddedChannel channel = new EmbeddedChannel(new AbsIntegerEncoder());
21.         // 将buf数据写入出站EmbeddedChannel
22.         Assert.assertTrue(channel.writeOutbound(buf));
23.         // 标示EmbeddedChannel完成
24.         Assert.assertTrue(channel.finish());
25.         // 读取出站数据
26.         ByteBuffer output = (ByteBuffer) channel.readOutbound();
27.         for (int i = 1; i < 10; i++) {
28.             Assert.assertEquals(i, output.readInt());
29.         }
30.         Assert.assertFalse(output.isReadable());
31.         Assert.assertNull(channel.readOutbound());
32.     }
33.
34. }
```

10.3 测试异常处理

有时候传输的进站或出站数据不够，通常这种情况也需要处理，例如抛出一个异常。这可能是你错误的输入或处理大的资源或其他的异常导致。我们来写一个实现，如果输入字节超出限制长度就抛出TooLongFrameException，这样的功能一般用来防止资源耗尽。看下图：



上图显示帧的大小被限制为3字节，若输入的字节超过3字节，则超过的字节被丢弃并抛出TooLongFrameException。在ChannelPipeline中的其他ChannelHandler实现可以处理TooLongFrameException或者忽略异常。处理异常在ChannelHandler.exceptionCaught()方法中完成，ChannelHandler提供了一些具体的实现，看下面代码：

[java]

```
01. package netty.in.action;
02.
03. import java.util.List;
04.
05. import io.netty.buffer.ByteBuf;
06. import io.netty.channel.ChannelHandlerContext;
07. import io.netty.handler.codec.ByteToMessageDecoder;
08. import io.netty.handler.codec.TooLongFrameException;
09.
10. public class FrameChunkDecoder extends ByteToMessageDecoder {
11.
12.     // 限制大小
13.     private final int maxFrameSize;
14.
15.     public FrameChunkDecoder(int maxFrameSize) {
16.         this.maxFrameSize = maxFrameSize;
17.     }
18.
19.     @Override
20.     protected void decode(ChannelHandlerContext ctx, ByteBuf in,
21.         List<Object> out) throws Exception {
22.         // 获取可读字节数
23.         int readableBytes = in.readableBytes();
24.         // 若可读字节数大于限制值, 清空字节并抛出异常
25.         if (readableBytes > maxFrameSize) {
26.             in.clear();
27.             throw new TooLongFrameException();
28.         }
29.         // 读取ByteBuf并放到List中
30.         ByteBuf buf = in.readBytes(readableBytes);
31.         out.add(buf);
32.     }
33.
34. }
```

测试FrameChunkDecoder的代码如下：

[java]

```
01. package netty.in.action;
02.
03. import io.netty.buffer.ByteBuf;
04. import io.netty.buffer.Unpooled;
05. import io.netty.channel.embedded.EmbeddedChannel;
06. import io.netty.handler.codec.TooLongFrameException;
07.
08. import org.junit.Assert;
09. import org.junit.Test;
10.
11. public class FrameChunkDecoderTest {
12.
13.     @Test
14.     public void testFramesDecoded() {
15.         // 创建ByteBuf并填充9字节数据
16.         ByteBuf buf = Unpooled.buffer();
17.         for (int i = 0; i < 9; i++) {
18.             buf.writeByte(i);
19.         }
20.         // 复制一个ByteBuf
21.         ByteBuf input = buf.duplicate();
22.         // 创建EmbeddedChannel
23.         EmbeddedChannel channel = new EmbeddedChannel(new FrameChunkDecoder(3));
24.         // 读取2个字节写入入站通道
25.         Assert.assertTrue(channel.writeInbound(input.readBytes(2)));
26.         try {
27.             // 读取4个字节写入入站通道
28.             channel.writeInbound(input.readBytes(4));
29.             Assert.fail();
30.         } catch (TooLongFrameException e) {
31.
32.         }
33.         // 读取3个字节写入入站通道
34.         Assert.assertTrue(channel.writeInbound(input.readBytes(3)));
35.         // 标识完成
36.         Assert.assertTrue(channel.finish());
37.         // 从EmbeddedChannel入去入站数据
38.         Assert.assertEquals(buf.readBytes(2), channel.readInbound());
```



```
39. Assert.assertEquals(buf.skipBytes(4).readBytes(3),
40.     channel.readInbound());
41. }
42.
43. }
```

10.4 Summary

In this chapter you learned how you are be able to test your custom ChannelHandler and so make sure it works like you expected. Using the shown techniques you are now be able to make use of JUnit and so ultimately test your code as your are used to. Using the techniques shown in the chapter you will be able to guarantee a high quality of your code and also guard it from misbehavior.. In the next chapters we will focus on writing "real" applications on top of Netty and so show you how you can make real use of it. Even if the applications don't contain any test-code remember it is quite important to do so when you will write your next-gen application.

Netty In Action中文版 - 第十一章：WebSocket

- 目录
- [-]
- 1. WebSockets some background
- 2. 面临的挑战
- 3. 实现
 - 1. 处理http请求
 - 2. 处理WebSocket框架
 - 3. 初始化ChannelPipeline
- 4. 结合在一起使用
- 5. 给WebSocket加密
- 6. Summary

本章介绍

- WebSocket
- ChannelHandler, Decoder and Encoder
- 引导一个Netty基础程序
- 测试WebSocket

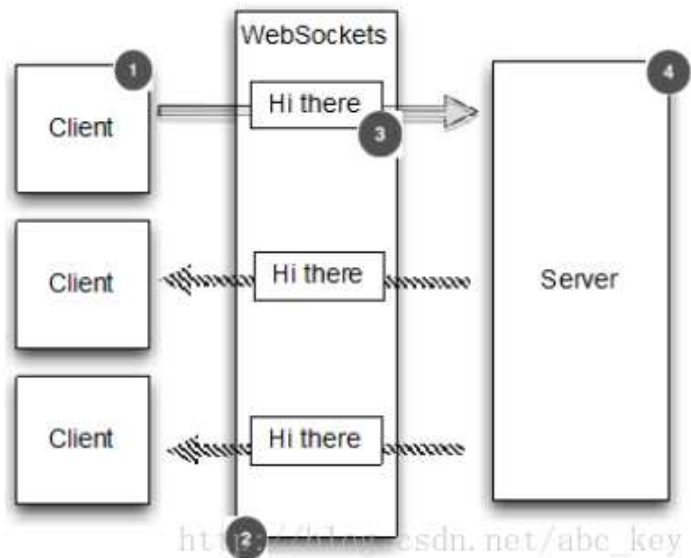
“real-time-web”实时web现在随处可见，很多的用户希望能从web站点实时获取信息。Netty支持WebSocket实现，并包含了不同的版本，我们可以非常容易的实现WebSocket应用。使用Netty附带的WebSocket，我们不需要关注协议内部实现，只需要使用Netty提供的一些简单的方法就可以实现。本章将通过的例子应用帮助你使用WebSocket并了解它是如何工作。

11.1 WebSockets some background

关于WebSocket的一些概念和背景，可以查询网上相关介绍。这里不赘述。

11.2 面临的挑战

要显示“real-time”支持的WebSocket，应用程序将显示如何使用Netty中的WebSocket实现一个在浏览器中进行聊天的IRC应用程序。你可能知道从Facebook可以发送文本消息到另一个人，在这里，我们将进一步了解其实现。在这个应用程序中，不同的用户可以同时交谈，非常像IRC(Internet Relay Chat，互联网中继聊天)。



上图显示的逻辑很简单：

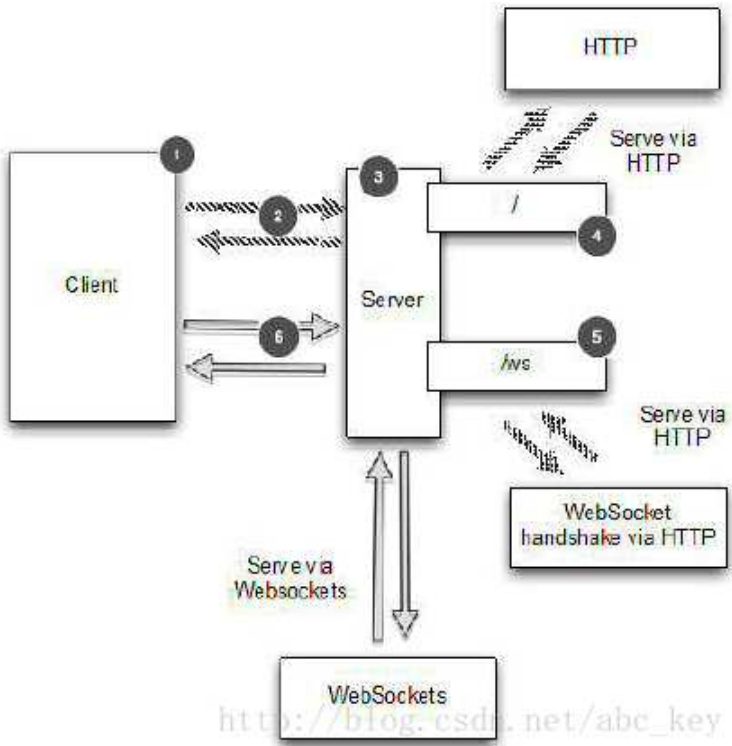
1. 一个客户端发送一条消息
2. 消息被广播到其他已连接的客户端

它的工作原理就像聊天室一样，在这里例子中，我们将编写服务器，然后使用浏览器作为客户端。带着这样的思路，我们将会很简单的实现它。

11.3 实现

WebSocket使用HTTP升级机制从一个普通的HTTP连接WebSocket，因为这个应用程序使用WebSocket总是开始于HTTP(s)，然后再升级。什么时候升级取决于应用程序本身。直接执行升级作为第一个操作一般是使用特定的url请求。

在这里，如果url的结尾以ws结束，我们将只会升级到WebSocket，否则服务器将发送一个网页给客户端。升级后的连接将通过WebSocket传输所有数据。逻辑图如下：



11.3.1 处理http请求

服务器将作为一种混合式以允许同时处理http和websocket，所以服务器还需要html页面，html用来充当客户端角色，连接服务器并交互消息。因此，如果客户端不发送/ws的uri，我们需要写一个ChannelInboundHandler用来处理FullHttpRequest。看下面代码：

```
[java]
01. package netty.in.action;
02.
03. import io.netty.channel.ChannelFuture;
04. import io.netty.channel.ChannelFutureListener;
05. import io.netty.channel.ChannelHandlerContext;
06. import io.netty.channel.DefaultFileRegion;
07. import io.netty.channel.SimpleChannelInboundHandler;
08. import io.netty.handler.codec.http.DefaultFullHttpResponse;
09. import io.netty.handler.codec.http.DefaultHttpResponse;
10. import io.netty.handler.codec.http.FullHttpRequest;
11. import io.netty.handler.codec.http.FullHttpResponse;
12. import io.netty.handler.codec.http.HttpHeaders;
13. import io.netty.handler.codec.http.HttpResponse;
14. import io.netty.handler.codec.http.HttpResponseStatus;
15. import io.netty.handler.codec.http.HttpVersion;
16. import io.netty.handler.codec.http.LastHttpContent;
17. import io.netty.handler.ssl.SslHandler;
18. import io.netty.handler.stream.ChunkedNioFile;
19.
20. import java.io.RandomAccessFile;
21.
22. /**
23.  * WebSocket，处理http请求
24.  *
25.  * @author c.k
26.  *
27.  */
28. public class HttpRequestHandler extends
29.     SimpleChannelInboundHandler<FullHttpRequest> {
30.     // websocket 标识
31.     private final String wsUri;
32.
33.     public HttpRequestHandler(String wsUri) {
34.         this.wsUri = wsUri;
35.     }
36.
37.     @Override
38.     protected void channelRead0(ChannelHandlerContext ctx, FullHttpRequest msg)
39.         throws Exception {
40.         // 如果是websocket 请求，请求地址uri 等于wsuri
41.         if (wsUri.equalsIgnoreCase(msg.getUri())) {
42.             // 将消息转发到下一个ChannelHandler
43.             ctx.fireChannelRead(msg.retain());
44.         } else { // 如果不是websocket 请求
45.             if (HttpHeaders.is100ContinueExpected(msg)) {
46.                 // 如果HTTP请求头部包含Expect: 100-continue,
47.                 // 则响应请求
48.                 FullHttpResponse response = new DefaultFullHttpResponse(
49.                     HttpVersion.HTTP_1_1, HttpResponseStatus.CONTINUE);
```

```
50.         ctx.writeAndFlush(response);
51.     }
52.     // 获取index.html 的内容响应给客户端
53.     RandomAccessFile file = new RandomAccessFile(
54.         System.getProperty("user.dir") + "/index.html", "r");
55.     HttpResponse response = new DefaultHttpResponse(
56.         msg.getProtocolVersion(), HttpResponseStatus.OK);
57.     response.headers().set(HttpHeaderNames.CONTENT_TYPE,
58.         "text/html; charset=UTF-8");
59.     boolean keepAlive = HttpHeaders.isKeepAlive(msg);
60.     // 如果http请求保持活跃, 设置http请求头部信息
61.     // 并响应请求
62.     if (keepAlive) {
63.         response.headers().set(HttpHeaderNames.CONTENT_LENGTH,
64.             file.length());
65.         response.headers().set(HttpHeaderNames.CONNECTION,
66.             HttpHeaders.Values.KEEP_ALIVE);
67.     }
68.     ctx.write(response);
69.     // 如果不是https 请求, 将index.html 内容写入通道
70.     if (ctx.pipeline().get(SslHandler.class) == null) {
71.         ctx.write(new DefaultFileRegion(file.getChannel(), 0, file
72.             .length()));
73.     } else {
74.         ctx.write(new ChunkedNioFile(file.getChannel()));
75.     }
76.     // 标识响应内容结束并刷新通道
77.     ChannelFuture future = ctx
78.         .writeAndFlush(LastHttpContent.EMPTY_LAST_CONTENT);
79.     if (!keepAlive) {
80.         // 如果http请求不活跃, 关闭http连接
81.         future.addListener(ChannelFutureListener.CLOSE);
82.     }
83.     file.close();
84. }
85. }
86.
87. @Override
88. public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)
89.     throws Exception {
90.     cause.printStackTrace();
91.     ctx.close();
92. }
93. }
```

11.3.2 处理WebSocket框架

WebSocket支持6种不同框架，如下图：

Name	Description
BinaryWebSocketFrame	WebSocketFrame that contains binary data
TextWebSocketFrame	WebSocketFrame that contains text data
ContinuationWebSocketFrame	WebSocketFrame that contains text or binary data that belongs to a previous BinaryWebSocketFrame or TextWebSocketFrame
CloseWebSocketFrame	WebSocketFrame that represent a CLOSE request and contains close status code and a phrase
PingWebSocketFrame	WebSocketFrame which request the send of a PongWebSocketFrame
PongWebSocketFrame	WebSocketFrame which is send as response of a PingWebSocketFrame

我们的程序只需要使用下面4个框架：

- CloseWebSocketFrame
- PingWebSocketFrame
- PongWebSocketFrame
- TextWebSocketFrame

我们只需要显示处理TextWebSocketFrame，其他的会自动由WebSocketServerProtocolHandler处理，看下面代码：

[java]  

```
01. package netty.in.action;
02.
03. import io.netty.channel.ChannelHandlerContext;
04. import io.netty.channel.SimpleChannelInboundHandler;
05. import io.netty.channel.group.ChannelGroup;
06. import io.netty.handler.codec.http.websocketx.TextWebSocketFrame;
07. import io.netty.handler.codec.http.websocketx.WebSocketServerProtocolHandler;
08.
09. /**
10.  * WebSocket，处理消息
11.  * @author c.k
12.  *
13.  */
14. public class TextWebSocketFrameHandler extends
15.     SimpleChannelInboundHandler<TextWebSocketFrame> {
16.     private final ChannelGroup group;
17.
18.     public TextWebSocketFrameHandler(ChannelGroup group) {
19.         this.group = group;
20.     }
21.
22.     @Override
23.     public void userEventTriggered(ChannelHandlerContext ctx, Object evt)
24.         throws Exception {
25.         // 如果WebSocket握手完成
26.         if (evt == WebSocketServerProtocolHandler.ServerHandshakeStateEvent.HANDSHAKE_COMPLETE) {
27.             // 删除ChannelPipeline中的HttpRequestHandler
28.             ctx.pipeline().remove(HttpRequestHandler.class);
29.             // 写一个消息到ChannelGroup
30.             group.writeAndFlush(new TextWebSocketFrame("Client " + ctx.channel()
31.                 + " joined"));
32.             // 将Channel添加到ChannelGroup
33.             group.add(ctx.channel());
34.         } else {
35.             super.userEventTriggered(ctx, evt);
36.         }
37.     }
38.
39.     @Override
40.     protected void channelRead0(ChannelHandlerContext ctx, TextWebSocketFrame msg)
41.         throws Exception {
42.         // 将接收的消息通过ChannelGroup转发到所以已连接的客户端
43.         group.writeAndFlush(msg.retain());
44.     }
45. }
```

11.3.3 初始化ChannelPipeline

看下面代码：

[java]  

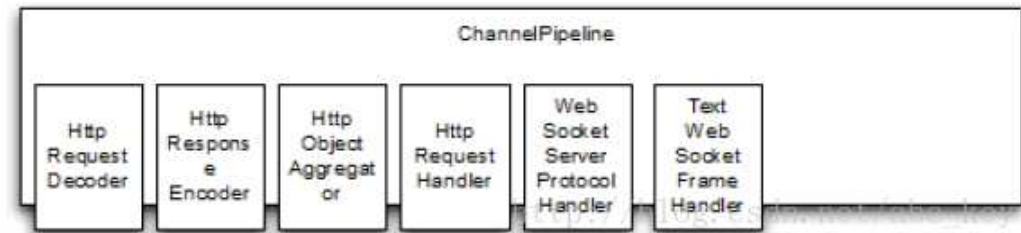
```
01. package netty.in.action;
02.
03. import io.netty.channel.Channel;
04. import io.netty.channel.ChannelInitializer;
05. import io.netty.channel.ChannelPipeline;
06. import io.netty.channel.group.ChannelGroup;
07. import io.netty.handler.codec.http.HttpObjectAggregator;
08. import io.netty.handler.codec.http.HttpServerCodec;
09. import io.netty.handler.codec.http.websocketx.WebSocketServerProtocolHandler;
10. import io.netty.handler.stream.ChunkedWriteHandler;
11.
12. /**
13.  * WebSocket，初始化ChannelHandler
14.  * @author c.k
15.  *
16.  */
17. public class ChatServerInitializer extends ChannelInitializer<Channel> {
18.     private final ChannelGroup group;
19.
20.     public ChatServerInitializer(ChannelGroup group){
21.         this.group = group;
22.     }
23.
24.     @Override
25.     protected void initChannel(Channel ch) throws Exception {
26.         ChannelPipeline pipeline = ch.pipeline();
```



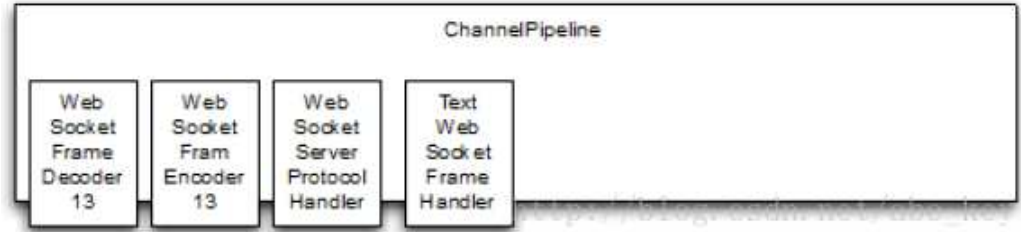
```
27. // 编解码http请求
28. pipeline.addLast(new HttpServerCodec());
29. // 写文件内容
30. pipeline.addLast(new ChunkedWriteHandler());
31. // 聚合解码HttpRequest/HttpContent/LastHttpContent 到FullHttpRequest
32. // 保证接收的HttpRequest的完整性
33. pipeline.addLast(new HttpObjectAggregator(64 * 1024));
34. // 处理FullHttpRequest
35. pipeline.addLast(new HttpRequestHandler("/ws"));
36. // 处理其他的WebSocketFrame
37. pipeline.addLast(new WebSocketServerProtocolHandler("/ws"));
38. // 处理TextWebSocketFrame
39. pipeline.addLast(new TextWebSocketFrameHandler(group));
40. }
41.
42. }
```

WebSocketServerProtocolHandler不仅处理Ping/Pong/CloseWebSocketFrame，还和它自己握手并帮助升级WebSocket。这是执行完成握手和成功修改ChannelPipeline，并且添加需要的编码器/解码器和删除不需要的ChannelHandler。

看下图：



ChannelPipeline通过ChannelInitializer的initChannel(...)方法完成初始化，完成握手后就会更改事情。一旦这样做了，WebSocketServerProtocolHandler将取代HttpRequestDecoder、WebSocketFrameDecoder13和HttpResponseEncoder、WebSocketFrameEncoder13。另外也要删除所有不需要的ChannelHandler已获得最佳性能。这些都是HttpObjectAggregator和HttpRequestHandler。下图显示ChannelPipeline握手完成：



我们甚至没注意到它，因为它是在底层执行的。以非常灵活的方式动态更新ChannelPipeline让单独的任务在不同的ChannelHandler中实现。

11.4 结合在一起使用

一如既往，我们要将它们结合在一起使用。使用Bootstrap引导服务器和设置正确的ChannelInitializer。看下面代码：

```
[java]

01. package netty.in.action;
02.
03. import io.netty.bootstrap.ServerBootstrap;
04. import io.netty.channel.Channel;
05. import io.netty.channel.ChannelFuture;
06. import io.netty.channel.ChannelInitializer;
07. import io.netty.channel.EventLoopGroup;
08. import io.netty.channel.group.ChannelGroup;
09. import io.netty.channel.group.DefaultChannelGroup;
10. import io.netty.channel.nio.NioEventLoopGroup;
11. import io.netty.channel.socket.nio.NioServerSocketChannel;
12. import io.netty.util.concurrent.ImmediateEventExecutor;
13.
14. import java.net.InetSocketAddress;
15.
16. /**
17.  * 访问地址: http://localhost:2048
18.  *
19.  * @author c.k
20.  *
21.  */
22. public class ChatServer {
23.
24.     private final ChannelGroup group = new DefaultChannelGroup(
25.         ImmediateEventExecutor.INSTANCE);
26.     private final EventLoopGroup workerGroup = new NioEventLoopGroup();
27.     private Channel channel;
```

```
28.
29.     public ChannelFuture start(InetSocketAddress address) {
30.         ServerBootstrap b = new ServerBootstrap();
31.         b.group(workerGroup).channel(NioServerSocketChannel.class)
32.             .childHandler(createInitializer(group));
33.         ChannelFuture f = b.bind(address).syncUninterruptibly();
34.         channel = f.channel();
35.         return f;
36.     }
37.
38.     public void destroy() {
39.         if (channel != null)
40.             channel.close();
41.         group.close();
42.         workerGroup.shutdownGracefully();
43.     }
44.
45.     protected ChannelInitializer<Channel> createInitializer(ChannelGroup group) {
46.         return new ChatServerInitializer(group);
47.     }
48.
49.     public static void main(String[] args) {
50.         final ChatServer server = new ChatServer();
51.         ChannelFuture f = server.start(new InetSocketAddress(2048));
52.         Runtime.getRuntime().addShutdownHook(new Thread() {
53.             @Override
54.             public void run() {
55.                 server.destroy();
56.             }
57.         });
58.         f.channel().closeFuture().syncUninterruptibly();
59.     }
60.
61. }
```

另外，需要将index.html文件放在项目根目录，index.html内容如下：

[html]  

```
01. <html>
02. <head>
03. <title>Web Socket Test</title>
04. </head>
05. <body>
06. <script type="text/javascript">
07. var socket;
08. if (!window.WebSocket) {
09.     window.WebSocket = window.MozWebSocket;
10. }
11. if (window.WebSocket) {
12.     socket = new WebSocket("ws://localhost:2048/ws");
13.     socket.onmessage = function(event) {
14.         var ta = document.getElementById('responseText');
15.         ta.value = ta.value + '\n' + event.data;
16.     };
17.     socket.onopen = function(event) {
18.         var ta = document.getElementById('responseText');
19.         ta.value = "Web Socket opened!";
20.     };
21.     socket.onclose = function(event) {
22.         var ta = document.getElementById('responseText');
23.         ta.value = ta.value + "Web Socket closed";
24.     };
25. } else {
26.     alert("Your browser does not support Web Socket.");
27. }
28.
29. function send(message) {
30.     if (!window.WebSocket) { return; }
31.     if (socket.readyState == WebSocket.OPEN) {
32.         socket.send(message);
33.     } else {
34.         alert("The socket is not open.");
35.     }
36. }
37. </script>
```

```
38.         <form onsubmit="return false;">
39.             <input type="text" name="message" value="Hello, World!"><input
40.                 type="button" value="Send Web Socket Data"
41.                 onclick="send(this.form.message.value)">
42.             <h3>Output </h3>
43.             <textarea id="responseText" style="width: 500px; height: 300px;"></textarea>
44.         </form>
45.     </body>
46. </html>
```

最后在浏览器中输入：<http://localhost:2048>，多开几个窗口就可以聊天了。

11.5 给WebSocket加密

上面的应用程序虽然工作的很好，但是在网络上收发消息存在很大的安全隐患，所以有必要对消息进行加密。添加这样一个加密的功能一般比较复杂，需要对代码有较大的改动。但是使用Netty就可以很容易的添加这样的功能，只需要将SslHandler加入到ChannelPipeline中就可以了。实际上还需要添加SslContext，但这不在本例子范围内。

首先我们创建一个用于添加加密Handler的handler初始化类，看下面代码：

```
[java]

01. package netty.in.action;
02.
03. import io.netty.channel.Channel;
04. import io.netty.channel.group.ChannelGroup;
05. import io.netty.handler.ssl.SslHandler;
06.
07. import javax.net.ssl.SSLContext;
08. import javax.net.ssl.SSLEngine;
09.
10. public class SecureChatServerInitializer extends ChatServerInitializer {
11.     private final SSLContext context;
12.
13.     public SecureChatServerInitializer(ChannelGroup group, SSLContext context) {
14.         super(group);
15.         this.context = context;
16.     }
17.
18.     @Override
19.     protected void initChannel(Channel ch) throws Exception {
20.         super.initChannel(ch);
21.         SSLEngine engine = context.createSSLEngine();
22.         engine.setUseClientMode(false);
23.         ch.pipeline().addFirst(new SslHandler(engine));
24.     }
25. }
```

最后我们创建一个用于引导配置的类，看下面代码：

```
[java]

01. package netty.in.action;
02.
03. import io.netty.channel.Channel;
04. import io.netty.channel.ChannelFuture;
05. import io.netty.channel.ChannelInitializer;
06. import io.netty.channel.group.ChannelGroup;
07. import java.net.InetSocketAddress;
08. import javax.net.ssl.SSLContext;
09.
10. /**
11.  * 访问地址: https://localhost:4096
12.  *
13.  * @author c.k
14.  *
15.  */
16. public class SecureChatServer extends ChatServer {
17.     private final SSLContext context;
18.
19.     public SecureChatServer(SSLContext context) {
20.         this.context = context;
21.     }
22.
23.     @Override
24.     protected ChannelInitializer<Channel> createInitializer(ChannelGroup group) {
25.         return new SecureChatServerInitializer(group, context);
26.     }
27. }
```

```
28.  /**
29.   * 获取SSLContext 需要相关的keystore文件，这里没有 关于HTTPS可以查阅相关资料，这里只介绍在Netty中如何使用
30.   *
31.   * @return
32.   */
33. private static SSLContext getSslContext() {
34.     return null;
35. }
36.
37. public static void main(String[] args) {
38.     SSLContext context = getSslContext();
39.     final SecureChatServer server = new SecureChatServer(context);
40.     ChannelFuture future = server.start(new InetSocketAddress(4096));
41.     Runtime.getRuntime().addShutdownHook(new Thread() {
42.         @Override
43.         public void run() {
44.             server.destroy();
45.         }
46.     });
47.     future.channel().closeFuture().syncUninterruptibly();
48. }
49. }
```

11.6 Summary

第十二章：SPDY

目录

- 1. SPDY概念及背景
- 2. 本例子流程图
- 3. Netty中使用SPDY
- 4. Summary

本章我将不会直接翻译Netty in Action书中的原文，感觉原书中本章讲的很多废话，我翻译起来也吃力。所以，本章内容我会根据其资料和个人理解来讲述。

12.1 SPDY概念及背景

SPDY 是 Google 开发的基于传输控制协议 (TCP) 的应用层协议，开发组正在推动 SPDY 成为正式标准（现为互联网草案）。SPDY 协议旨在通过压缩、多路复用和优先级来缩短网页的加载时间和提高安全性。（SPDY 是 Speedy 的呢音，意思是更快）。

为什么需要SPDY? SPDY 协议只是在性能上对 HTTP 做了很大的优化, 其核心思想是尽量减少连接个数, 而对于 HTTP 的语义并没有做太大的修改。具体来说, SPDY 使用了 HTTP 的方法和页眉, 但是删除了一些头并重写了 HTTP 中管理连接和数据转移格式的部分, 所以基本上是兼容 HTTP 的。

Google 在 SPDY 白皮书里表示要向协议栈下面渗透并替换掉传输层协议（TCP），但是因为这样无论是部署起来还是实现起来暂时相当困难，因此 Google 准备先对应用层协议 HTTP 进行改进，先在 SSL 之上增加一个会话层来实现 SPDY 协议，而 HTTP 的 GET 和 POST 消息格式保持不变，即现有的所有服务端应用均不用做任何修改。因此在目前，SPDY 的目的是为了加强 HTTP，是对 HTTP 一个更好的实现和支持。至于未来 SPDY 得到广泛应用后会不会演一出狸猫换太子，替换掉 HTTP 并彻底颠覆整个 Internet 就是 Google 的事情了。

距离万维网之父蒂姆·伯纳斯·李发明并推动 HTTP 成为如今互联网最流行的协议已经过去十几年了（现用 HTTP 1.1 规范也停滞了 13 年了），随着现在 WEB 技术的飞速发展尤其是 HTML5 的不断演进，包括 WebSockets 协议的出现以及当前网络环境的改变、传输内容的变化，当初的 HTTP 规范已经逐渐无法满足人们的需要了，HTTP 需要进一步发展，因此 HTTPbis 工作组已经被组建并被授权考虑 HTTP 2.0，希望能解决掉目前 HTTP 所带来的诸多限制。而 SPDY 正是 Google 在 HTTP 即将从 1.1 跨越到 2.0 之际推出的试图成为下一代互联网通信的协议，长期以来一直被认为是 HTTP 2.0 唯一可行选择。

SPDY相比HTTP有如下优点:

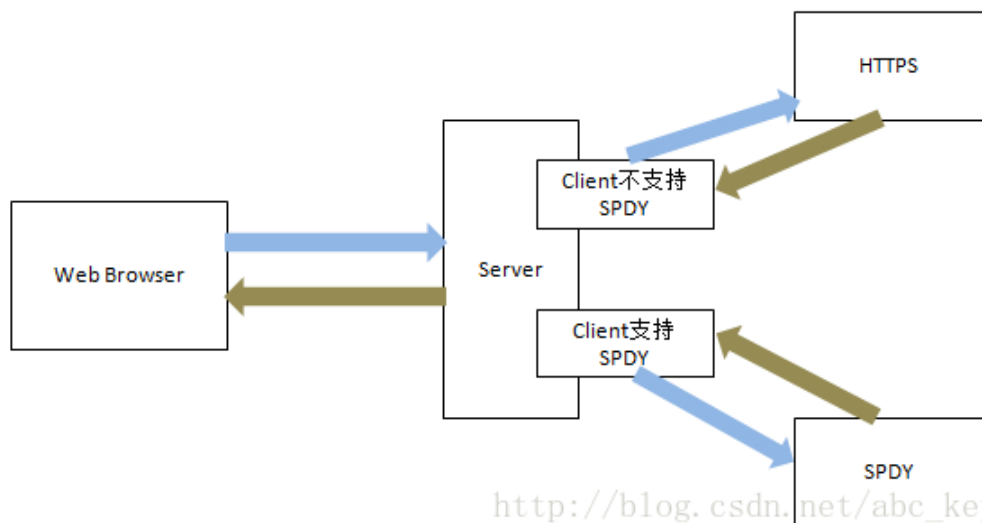
1. SPDY多路复用，请求优化；而HTTP单路连接，请求低效
2. SPDY支持服务器推送技术；而HTTP只允许由客户端主动发起请求
3. SPDY压缩了HTTP头信息，节省了传输数据的带宽流量；而HTTP头冗余，同一个会话会反复送头信息
4. SPDY强制使用SSL传输协议，全部请求SSL加密后，信息传输更安全

谷歌表示，引入SPDY协议后，在实验室测试中页面加载速度比原先快64%。

支持SPDY协议的浏览器:

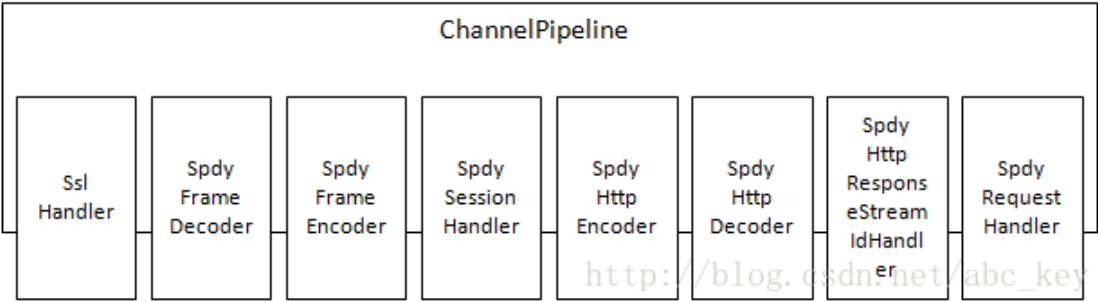
- Google Chrome 19+和Chromium 19+
- Mozilla Firefox 11+, 从13开始默认支持
- Opera 12.10+
- Internet Explorer 11+

12.2 本例子流程图

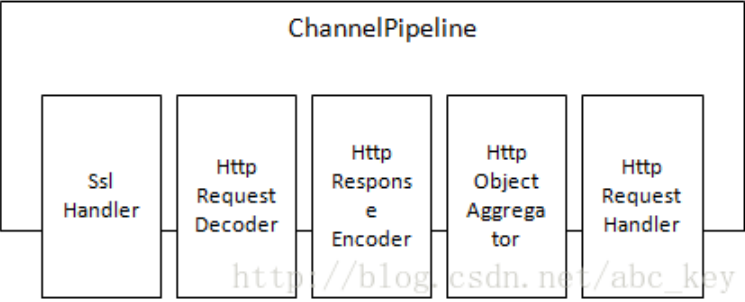


12.3 Netty中使用SPDY

支持SPDY的ChannelPipeline如下图:



不支持SPDY的ChannelPipeline如下图：



例子代码如下：

```
[java]
01. package netty.in.action.spdy;
02.
03. import java.util.Arrays;
04. import java.util.Collections;
05. import java.util.List;
06.
07. import org.eclipse.jetty.npn.NextProtoNego.ServerProvider;
08.
09. public class DefaultServerProvider implements ServerProvider {
10.
11.     private static final List<String> PROTOCOLS = Collections.unmodifiableList(Arrays
12.         .asList("spdy/3.1", "http/1.1", "http/1.0", "Unknown"));
13.
14.     private String protocol;
15.
16.     public String getSelectedProtocol() {
17.         return protocol;
18.     }
19.
20.     @Override
21.     public void protocolSelected(String arg0) {
22.         this.protocol = arg0;
23.     }
24.
25.     @Override
26.     public List<String> protocols() {
27.         return PROTOCOLS;
28.     }
29.
30.     @Override
31.     public void unsupported() {
32.         protocol = "http/1.1";
33.     }
34.
35. }
```

```
[java]
01. package netty.in.action.spdy;
02.
03. import io.netty.channel.ChannelFuture;
04. import io.netty.channel.ChannelFutureListener;
05. import io.netty.channel.ChannelHandlerContext;
06. import io.netty.channel.SimpleChannelInboundHandler;
07. import io.netty.handler.codec.http.DefaultFullHttpRequest;
08. import io.netty.handler.codec.http.FullHttpRequest;
09. import io.netty.handler.codec.http.FullHttpResponse;
10. import io.netty.handler.codec.http.HttpHeaders;
11. import io.netty.handler.codec.http.HttpResponseStatus;
12. import io.netty.handler.codec.http.HttpVersion;
13. import io.netty.util.CharsetUtil;
14.
15. public class HttpRequestHandler extends SimpleChannelInboundHandler<FullHttpRequest> {
16.
```

```
17. @Override
18. protected void channelRead0(ChannelHandlerContext ctx, FullHttpRequest request)
19.     throws Exception {
20.     if (HttpHeaders.is100ContinueExpected(request)) {
21.         send100Continue(ctx);
22.     }
23.     FullHttpResponse response = new DefaultFullHttpResponse(
24.         request.getProtocolVersion(), HttpResponseStatus.OK);
25.     response.content().writeBytes(getContent().getBytes(CharsetUtil.UTF_8));
26.     response.headers().set(HttpHeaders.Names.CONTENT_TYPE,
27.         "text/plain; charset=UTF-8");
28.     boolean keepAlive = HttpHeaders.isKeepAlive(request);
29.     if (keepAlive) {
30.         response.headers().set(HttpHeaders.Names.CONTENT_LENGTH,
31.             response.content().readableBytes());
32.         response.headers().set(HttpHeaders.Names.CONNECTION,
33.             HttpHeaders.Values.KEEP_ALIVE);
34.     }
35.     ChannelFuture future = ctx.writeAndFlush(response);
36.     if (!keepAlive) {
37.         future.addListener(ChannelFutureListener.CLOSE);
38.     }
39. }
40.
41. private static void send100Continue(ChannelHandlerContext ctx) {
42.     FullHttpResponse response = new DefaultFullHttpResponse(HttpVersion.HTTP_1_1,
43.         HttpResponseStatus.CONTINUE);
44.     ctx.writeAndFlush(response);
45. }
46.
47. protected String getContent() {
48.     return "This content is transmitted via HTTP\r\n";
49. }
50.
51. @Override
52. public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)
53.     throws Exception {
54.     cause.printStackTrace();
55.     ctx.close();
56. }
57. }
```

[java]  

```
01. package netty.in.action.spdy;
02.
03. public class SpdyRequestHandler extends HttpRequestHandler {
04.
05.     @Override
06.     protected String getContent() {
07.         return "This content is transmitted via SPDY\r\n";
08.     }
09.
10. }
```

[java]  

```
01. package netty.in.action.spdy;
02.
03. import io.netty.channel.ChannelInboundHandler;
04. import io.netty.handler.codec.spdy.SpdyOrHttpChooser;
05.
06. import javax.net.ssl.SSLEngine;
07.
08. import org.eclipse.jetty.npn.NextProtocolNego;
09.
10. public class DefaultSpdyOrHttpChooser extends SpdyOrHttpChooser {
11.
12.     protected DefaultSpdyOrHttpChooser(int maxSpdyContentLength, int maxHttpContentLength) {
13.         super(maxSpdyContentLength, maxHttpContentLength);
14.     }
15.
16.     @Override
17.     protected SelectedProtocol getProtocol(SSLEngine engine) {
18.         DefaultServerProvider provider = (DefaultServerProvider) NextProtocolNego
19.             .get(engine);
20.         String protocol = provider.getSelectedProtocol();
21.         if (protocol == null) {
22.             return SelectedProtocol.UNKNOWN;
```

```
23.     }
24.     switch (protocol) {
25.     case "spdy/3.1":
26.         return SelectedProtocol.SPDY_3_1;
27.     case "http/1.0":
28.     case "http/1.1":
29.         return SelectedProtocol.HTTP_1_1;
30.     default:
31.         return SelectedProtocol.UNKNOWN;
32.     }
33. }
34.
35. @Override
36. protected ChannelInboundHandler createHttpRequestHandlerForHttp() {
37.     return new HttpRequestHandler();
38. }
39.
40. @Override
41. protected ChannelInboundHandler createHttpRequestHandlerForSpdy() {
42.     return new SpdyRequestHandler();
43. }
44.
45. }
```

[java]  

```
01. package netty.in.action.spdy;
02.
03. import io.netty.channel.Channel;
04. import io.netty.channel.ChannelInitializer;
05. import io.netty.channel.ChannelPipeline;
06. import io.netty.handler.ssl.SslHandler;
07.
08. import javax.net.ssl.SSLContext;
09. import javax.net.ssl.SSLEngine;
10.
11. import org.eclipse.jetty.npn.NextProtoNego;
12.
13. public class SpdyChannelInitializer extends ChannelInitializer<Channel> {
14.     private final SSLContext context;
15.
16.     public SpdyChannelInitializer(SSLContext context) {
17.         this.context = context;
18.     }
19.
20.     @Override
21.     protected void initChannel(Channel ch) throws Exception {
22.         ChannelPipeline pipeline = ch.pipeline();
23.         SSLEngine engine = context.createSSLEngine();
24.         engine.setUseClientMode(false);
25.         NextProtoNego.put(engine, new DefaultServerProvider());
26.         NextProtoNego.debug = true;
27.         pipeline.addLast("sslHandler", new SslHandler(engine));
28.         pipeline.addLast("chooser",
29.             new DefaultSpdyHttpRequestChooser(1024 * 1024, 1024 * 1024));
30.     }
31.
32. }
```

[java]  

```
01. package netty.in.action.spdy;
02.
03. import io.netty.bootstrap.ServerBootstrap;
04. import io.netty.channel.Channel;
05. import io.netty.channel.ChannelFuture;
06. import io.netty.channel.nio.NioEventLoopGroup;
07. import io.netty.channel.socket.nio.NioServerSocketChannel;
08. import io.netty.example.securechat.SecureChatSslContextFactory;
09.
10. import java.net.InetSocketAddress;
11.
12. import javax.net.ssl.SSLContext;
13.
14. public class SpdyServer {
15.
16.     private final NioEventLoopGroup group = new NioEventLoopGroup();
17.     private final SSLContext context;
18.     private Channel channel;
```

```
19.
20.     public SpdyServer(SSLContext context) {
21.         this.context = context;
22.     }
23.
24.     public ChannelFuture start(InetSocketAddress address) {
25.         ServerBootstrap bootstrap = new ServerBootstrap();
26.         bootstrap.group(group).channel(NioServerSocketChannel.class)
27.             .childHandler(new SpdyChannelInitializer(context));
28.         ChannelFuture future = bootstrap.bind(address);
29.         future.syncUninterruptibly();
30.         channel = future.channel();
31.         return future;
32.     }
33.
34.     public void destroy() {
35.         if (channel != null) {
36.             channel.close();
37.         }
38.         group.shutdownGracefully();
39.     }
40.
41.     public static void main(String[] args) {
42.         SSLContext context = SecureChatSSLContextFactory.getServerContext();
43.         final SpdyServer endpoint = new SpdyServer(context);
44.         ChannelFuture future = endpoint.start(new InetSocketAddress(4096));
45.         Runtime.getRuntime().addShutdownHook(new Thread() {
46.             @Override
47.             public void run() {
48.                 endpoint.destroy();
49.             }
50.         });
51.         future.channel().closeFuture().syncUninterruptibly();
52.     }
53.
54. }
```

使用SSL需要使用到SSLContext，下面代买是获取SSLContext对象：

```
[java]
01.  /*
02.   * Copyright 2012 The Netty Project
03.   *
04.   * The Netty Project licenses this file to you under the Apache License,
05.   * version 2.0 (the "License"); you may not use this file except in compliance
06.   * with the License. You may obtain a copy of the License at:
07.   *
08.   * http://www.apache.org/licenses/LICENSE-2.0
09.   *
10.   * Unless required by applicable law or agreed to in writing, software
11.   * distributed under the License is distributed on an "AS IS" BASIS, WITHOUT
12.   * WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
13.   * License for the specific language governing permissions and limitations
14.   * under the License.
15.   */
16. package netty.in.action.spdy;
17.
18. import javax.net.ssl.ManagerFactoryParameters;
19. import javax.net.ssl.TrustManager;
20. import javax.net.ssl.TrustManagerFactorySpi;
21. import javax.net.ssl.X509TrustManager;
22. import java.security.InvalidAlgorithmParameterException;
23. import java.security.KeyStore;
24. import java.security.KeyStoreException;
25. import java.security.cert.X509Certificate;
26.
27. /**
28.  * Bogus {@link TrustManagerFactorySpi} which accepts any certificate
29.  * even if it is invalid.
30.  */
31. public class SecureChatTrustManagerFactory extends TrustManagerFactorySpi {
32.
33.     private static final TrustManager DUMMY_TRUST_MANAGER = new X509TrustManager() {
34.         @Override
35.         public X509Certificate[] getAcceptedIssuers() {
36.             return new X509Certificate[0];
37.         }
38.     }
```

```

39.         @Override
40.         public void checkClientTrusted(X509Certificate[] chain, String authType) {
41.             // Always trust - it is an example.
42.             // You should do something in the real world.
43.             // You will reach here only if you enabled client certificate auth,
44.             // as described in SecureChatSslContextFactory.
45.             System.err.println(
46.                 "UNKNOWN CLIENT CERTIFICATE: " + chain[0].getSubjectDN());
47.         }
48.
49.         @Override
50.         public void checkServerTrusted(X509Certificate[] chain, String authType) {
51.             // Always trust - it is an example.
52.             // You should do something in the real world.
53.             System.err.println(
54.                 "UNKNOWN SERVER CERTIFICATE: " + chain[0].getSubjectDN());
55.         }
56.     };
57.
58.     public static TrustManager[] getTrustManagers() {
59.         return new TrustManager[] { DUMMY_TRUST_MANAGER };
60.     }
61.
62.     @Override
63.     protected TrustManager[] engineGetTrustManagers() {
64.         return getTrustManagers();
65.     }
66.
67.     @Override
68.     protected void engineInit(KeyStore keystore) throws KeyStoreException {
69.         // Unused
70.     }
71.
72.     @Override
73.     protected void engineInit(ManagerFactoryParameters managerFactoryParameters)
74.         throws InvalidAlgorithmParameterException {
75.         // Unused
76.     }
77. }

```

```

[java]
01.  /*
02.   * Copyright 2012 The Netty Project
03.   *
04.   * The Netty Project licenses this file to you under the Apache License,
05.   * version 2.0 (the "License"); you may not use this file except in compliance
06.   * with the License. You may obtain a copy of the License at:
07.   *
08.   * http://www.apache.org/licenses/LICENSE-2.0
09.   *
10.   * Unless required by applicable law or agreed to in writing, software
11.   * distributed under the License is distributed on an "AS IS" BASIS, WITHOUT
12.   * WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
13.   * License for the specific language governing permissions and limitations
14.   * under the License.
15.   */
16. package netty.in.action.spdy;
17.
18. import java.io.ByteArrayInputStream;
19. import java.io.InputStream;
20.
21. /**
22.  * A bogus key store which provides all the required information to
23.  * create an example SSL connection.
24.  *
25.  * To generate a bogus key store:
26.  * <pre>
27.  * keytool -genkey -alias securechat -keysize 2048 -validity 36500
28.  *          -keyalg RSA -dname "CN=securechat"
29.  *          -keypass secret -storepass secret
30.  *          -keystore cert.jks
31.  * </pre>
32.  */
33. public final class SecureChatKeyStore {
34.     private static final short[] DATA = {
35.         0xfe, 0xed, 0xfe, 0xed, 0x00, 0x00, 0x00, 0x02,
36.         0x00, 0x00, 0x00, 0x02, 0x00, 0x00, 0x00, 0x01,
37.         0x00, 0x07, 0x65, 0x78, 0x61, 0x6d, 0x70, 0x6c,

```


38.	0x65, 0x00, 0x00, 0x01, 0x1a, 0x9f, 0x57, 0xa5,
39.	0x27, 0x00, 0x00, 0x01, 0x9a, 0x30, 0x82, 0x01,
40.	0x96, 0x30, 0x0e, 0x06, 0x0a, 0x2b, 0x06, 0x01,
41.	0x04, 0x01, 0x2a, 0x02, 0x11, 0x01, 0x01, 0x05,
42.	0x00, 0x04, 0x82, 0x01, 0x82, 0x48, 0x6d, 0xcf,
43.	0x16, 0xb5, 0x50, 0x95, 0x36, 0xbf, 0x47, 0x27,
44.	0x50, 0x58, 0x0d, 0xa2, 0x52, 0x7e, 0x25, 0xab,
45.	0x14, 0x1a, 0x26, 0x5e, 0x2d, 0x8a, 0x23, 0x90,
46.	0x60, 0x7f, 0x12, 0x20, 0x56, 0xd1, 0x43, 0xa2,
47.	0x6b, 0x47, 0x5d, 0xed, 0x9d, 0xd4, 0xe5, 0x83,
48.	0x28, 0x89, 0xc2, 0x16, 0x4c, 0x76, 0x06, 0xad,
49.	0x8e, 0x8c, 0x29, 0x1a, 0x9b, 0x0f, 0xdd, 0x60,
50.	0x4b, 0xb4, 0x62, 0x82, 0x9e, 0x4a, 0x63, 0x83,
51.	0x2e, 0xd2, 0x43, 0x78, 0xc2, 0x32, 0x1f, 0x60,
52.	0xa9, 0x8a, 0x7f, 0x0f, 0x7c, 0xa6, 0x1d, 0xe6,
53.	0x92, 0x9e, 0x52, 0xc7, 0x7d, 0xbb, 0x35, 0x3b,
54.	0xaa, 0x89, 0x73, 0x4c, 0xfb, 0x99, 0x54, 0x97,
55.	0x99, 0x28, 0x6e, 0x66, 0x5b, 0xf7, 0x9b, 0x7e,
56.	0x6d, 0x8a, 0x2f, 0xfa, 0xc3, 0x1e, 0x71, 0xb9,
57.	0xbd, 0x8f, 0xc5, 0x63, 0x25, 0x31, 0x20, 0x02,
58.	0xff, 0x02, 0xf0, 0xc9, 0x2c, 0xdd, 0x3a, 0x10,
59.	0x30, 0xab, 0xe5, 0xad, 0x3d, 0x1a, 0x82, 0x77,
60.	0x46, 0xed, 0x03, 0x38, 0xa4, 0x73, 0x6d, 0x36,
61.	0x36, 0x33, 0x70, 0xb2, 0x63, 0x20, 0xca, 0x03,
62.	0xbf, 0x5a, 0xf4, 0x7c, 0x35, 0xf0, 0x63, 0x1a,
63.	0x12, 0x33, 0x12, 0x58, 0xd9, 0xa2, 0x63, 0x6b,
64.	0x63, 0x82, 0x41, 0x65, 0x70, 0x37, 0x4b, 0x99,
65.	0x04, 0x9f, 0xdd, 0x5e, 0x07, 0x01, 0x95, 0x9f,
66.	0x36, 0xe8, 0xc3, 0x66, 0x2a, 0x21, 0x69, 0x68,
67.	0x40, 0xe6, 0xbc, 0xbb, 0x85, 0x81, 0x21, 0x13,
68.	0xe6, 0xa4, 0xcf, 0xd3, 0x67, 0xe3, 0xfd, 0x75,
69.	0xf0, 0xdf, 0x83, 0xe0, 0xc5, 0x36, 0x09, 0xac,
70.	0x1b, 0xd4, 0xf7, 0x2a, 0x23, 0x57, 0x1c, 0x5c,
71.	0x0f, 0xf4, 0xcf, 0xa2, 0xcf, 0xf5, 0xbd, 0x9c,
72.	0x69, 0x98, 0x78, 0x3a, 0x25, 0xe4, 0xfd, 0x85,
73.	0x11, 0xcc, 0x7d, 0xef, 0xeb, 0x74, 0x60, 0xb1,
74.	0xb7, 0xfb, 0x1f, 0x0e, 0x62, 0xff, 0xfe, 0x09,
75.	0x0a, 0xc3, 0x80, 0x2f, 0x10, 0x49, 0x89, 0x78,
76.	0xd2, 0x08, 0xfa, 0x89, 0x22, 0x45, 0x91, 0x21,
77.	0xbc, 0x90, 0x3e, 0xad, 0xb3, 0x0a, 0xb4, 0x0e,
78.	0x1c, 0xa1, 0x93, 0x92, 0xd8, 0x72, 0x07, 0x54,
79.	0x60, 0xe7, 0x91, 0xfc, 0xd9, 0x3c, 0xe1, 0x6f,
80.	0x08, 0xe4, 0x56, 0xf6, 0x0b, 0xb0, 0x3c, 0x39,
81.	0x8a, 0x2d, 0x48, 0x44, 0x28, 0x13, 0xca, 0xe9,
82.	0xf7, 0xa3, 0xb6, 0x8a, 0x5f, 0x31, 0xa9, 0x72,
83.	0xf2, 0xde, 0x96, 0xf2, 0xb1, 0x53, 0xb1, 0x3e,
84.	0x24, 0x57, 0xfd, 0x18, 0x45, 0x1f, 0xc5, 0x33,
85.	0x1b, 0xa4, 0xe8, 0x21, 0xfa, 0x0e, 0xb2, 0xb9,
86.	0xcb, 0xc7, 0x07, 0x41, 0xdd, 0x2f, 0xb6, 0x6a,
87.	0x23, 0x18, 0xed, 0xc1, 0xef, 0xe2, 0x4b, 0xec,
88.	0xc9, 0xba, 0xfb, 0x46, 0x43, 0x90, 0xd7, 0xb5,
89.	0x68, 0x28, 0x31, 0x2b, 0x8d, 0xa8, 0x51, 0x63,
90.	0xf7, 0x53, 0x99, 0x19, 0x68, 0x85, 0x66, 0x00,
91.	0x00, 0x00, 0x01, 0x00, 0x05, 0x58, 0x2e, 0x35,
92.	0x30, 0x39, 0x00, 0x00, 0x02, 0x3a, 0x30, 0x82,
93.	0x02, 0x36, 0x30, 0x82, 0x01, 0xe0, 0xa0, 0x03,
94.	0x02, 0x01, 0x02, 0x02, 0x04, 0x48, 0x59, 0xf1,
95.	0x92, 0x30, 0x0d, 0x06, 0x09, 0x2a, 0x86, 0x48,
96.	0x86, 0xf7, 0x0d, 0x01, 0x01, 0x05, 0x05, 0x00,
97.	0x30, 0x81, 0xa0, 0x31, 0x0b, 0x30, 0x09, 0x06,
98.	0x03, 0x55, 0x04, 0x06, 0x13, 0x02, 0x4b, 0x52,
99.	0x31, 0x13, 0x30, 0x11, 0x06, 0x03, 0x55, 0x04,
100.	0x08, 0x13, 0x0a, 0x4b, 0x79, 0x75, 0x6e, 0x67,
101.	0x67, 0x69, 0x2d, 0x64, 0x6f, 0x31, 0x14, 0x30,
102.	0x12, 0x06, 0x03, 0x55, 0x04, 0x07, 0x13, 0x0b,
103.	0x53, 0x65, 0x6f, 0x6e, 0x67, 0x6e, 0x61, 0x6d,
104.	0x2d, 0x73, 0x69, 0x31, 0x1a, 0x30, 0x18, 0x06,
105.	0x03, 0x55, 0x04, 0x0a, 0x13, 0x11, 0x54, 0x68,
106.	0x65, 0x20, 0x4e, 0x65, 0x74, 0x74, 0x79, 0x20,
107.	0x50, 0x72, 0x6f, 0x6a, 0x65, 0x63, 0x74, 0x31,
108.	0x18, 0x30, 0x16, 0x06, 0x03, 0x55, 0x04, 0x0b,
109.	0x13, 0x0f, 0x45, 0x78, 0x61, 0x6d, 0x70, 0x6c,
110.	0x65, 0x20, 0x41, 0x75, 0x74, 0x68, 0x6f, 0x72,
111.	0x73, 0x31, 0x30, 0x30, 0x2e, 0x06, 0x03, 0x55,
112.	0x04, 0x03, 0x13, 0x27, 0x73, 0x65, 0x63, 0x75,
113.	0x72, 0x65, 0x63, 0x68, 0x61, 0x74, 0x2e, 0x65,
114.	0x78, 0x61, 0x6d, 0x70, 0x6c, 0x65, 0x2e, 0x6e,
115.	0x65, 0x74, 0x74, 0x79, 0x2e, 0x67, 0x6c, 0x65,
116.	0x61, 0x6d, 0x79, 0x6e, 0x6f, 0x64, 0x65, 0x2e,

117.	0x6e, 0x65, 0x74, 0x30, 0x20, 0x17, 0x0d, 0x30,
118.	0x38, 0x30, 0x36, 0x31, 0x39, 0x30, 0x35, 0x34,
119.	0x31, 0x33, 0x38, 0x5a, 0x18, 0x0f, 0x32, 0x31,
120.	0x38, 0x37, 0x31, 0x31, 0x32, 0x34, 0x30, 0x35,
121.	0x34, 0x31, 0x33, 0x38, 0x5a, 0x30, 0x81, 0xa0,
122.	0x31, 0x0b, 0x30, 0x09, 0x06, 0x03, 0x55, 0x04,
123.	0x06, 0x13, 0x02, 0x4b, 0x52, 0x31, 0x13, 0x30,
124.	0x11, 0x06, 0x03, 0x55, 0x04, 0x08, 0x13, 0x0a,
125.	0x4b, 0x79, 0x75, 0x6e, 0x67, 0x67, 0x69, 0x2d,
126.	0x64, 0x6f, 0x31, 0x14, 0x30, 0x12, 0x06, 0x03,
127.	0x55, 0x04, 0x07, 0x13, 0x0b, 0x53, 0x65, 0x6f,
128.	0x6e, 0x67, 0x6e, 0x61, 0x6d, 0x2d, 0x73, 0x69,
129.	0x31, 0x1a, 0x30, 0x18, 0x06, 0x03, 0x55, 0x04,
130.	0x0a, 0x13, 0x11, 0x54, 0x68, 0x65, 0x20, 0x4e,
131.	0x65, 0x74, 0x74, 0x79, 0x20, 0x50, 0x72, 0x6f,
132.	0x6a, 0x65, 0x63, 0x74, 0x31, 0x18, 0x30, 0x16,
133.	0x06, 0x03, 0x55, 0x04, 0x0b, 0x13, 0x0f, 0x45,
134.	0x78, 0x61, 0x6d, 0x70, 0x6c, 0x65, 0x20, 0x41,
135.	0x75, 0x74, 0x68, 0x6f, 0x72, 0x73, 0x31, 0x30,
136.	0x30, 0x2e, 0x06, 0x03, 0x55, 0x04, 0x03, 0x13,
137.	0x27, 0x73, 0x65, 0x63, 0x75, 0x72, 0x65, 0x63,
138.	0x68, 0x61, 0x74, 0x2e, 0x65, 0x78, 0x61, 0x6d,
139.	0x70, 0x6c, 0x65, 0x2e, 0x6e, 0x65, 0x74, 0x74,
140.	0x79, 0x2e, 0x67, 0x6c, 0x65, 0x61, 0x6d, 0x79,
141.	0x6e, 0x6f, 0x64, 0x65, 0x2e, 0x6e, 0x65, 0x74,
142.	0x30, 0x5c, 0x30, 0x0d, 0x06, 0x09, 0x2a, 0x86,
143.	0x48, 0x86, 0xf7, 0x0d, 0x01, 0x01, 0x01, 0x05,
144.	0x00, 0x03, 0x4b, 0x00, 0x30, 0x48, 0x02, 0x41,
145.	0x00, 0xc3, 0xe3, 0x5e, 0x41, 0xa7, 0x87, 0x11,
146.	0x00, 0x42, 0x2a, 0xb0, 0x4b, 0xed, 0xb2, 0xe0,
147.	0x23, 0xdb, 0xb1, 0x3d, 0x58, 0x97, 0x35, 0x60,
148.	0x0b, 0x82, 0x59, 0xd3, 0x00, 0xea, 0xd4, 0x61,
149.	0xb8, 0x79, 0x3f, 0xb6, 0x3c, 0x12, 0x05, 0x93,
150.	0x2e, 0x9a, 0x59, 0x68, 0x14, 0x77, 0x3a, 0xc8,
151.	0x50, 0x25, 0x57, 0xa4, 0x49, 0x18, 0x63, 0x41,
152.	0xf0, 0x2d, 0x28, 0xec, 0x06, 0xfb, 0xb4, 0x9f,
153.	0xbf, 0x02, 0x03, 0x01, 0x00, 0x01, 0x30, 0x0d,
154.	0x06, 0x09, 0x2a, 0x86, 0x48, 0x86, 0xf7, 0x0d,
155.	0x01, 0x01, 0x05, 0x05, 0x00, 0x03, 0x41, 0x00,
156.	0x65, 0x6c, 0x30, 0x01, 0xc2, 0x8e, 0x3e, 0xcb,
157.	0xb3, 0x77, 0x48, 0xe9, 0x66, 0x61, 0x9a, 0x40,
158.	0x86, 0xaf, 0xf6, 0x03, 0xeb, 0xba, 0x6a, 0xf2,
159.	0xfd, 0xe2, 0xaf, 0x36, 0x5e, 0x7b, 0xaa, 0x22,
160.	0x04, 0xdd, 0x2c, 0x20, 0xc4, 0xfc, 0xdd, 0xd0,
161.	0x82, 0x20, 0x1c, 0x3d, 0xd7, 0x9e, 0x5e, 0x5c,
162.	0x92, 0x5a, 0x76, 0x71, 0x28, 0xf5, 0x07, 0x7d,
163.	0xa2, 0x81, 0xba, 0x77, 0x9f, 0x2a, 0xd9, 0x44,
164.	0x00, 0x00, 0x00, 0x01, 0x00, 0x05, 0x6d, 0x79,
165.	0x6b, 0x65, 0x79, 0x00, 0x00, 0x01, 0x1a, 0x9f,
166.	0x5b, 0x56, 0xa0, 0x00, 0x00, 0x01, 0x99, 0x30,
167.	0x82, 0x01, 0x95, 0x30, 0x0e, 0x06, 0x0a, 0x2b,
168.	0x06, 0x01, 0x04, 0x01, 0x2a, 0x02, 0x11, 0x01,
169.	0x01, 0x05, 0x00, 0x04, 0x82, 0x01, 0x81, 0x29,
170.	0xa8, 0xb6, 0x08, 0x0c, 0x85, 0x75, 0x3e, 0xdd,
171.	0xb5, 0xe5, 0x1a, 0x87, 0x68, 0xd1, 0x90, 0x4b,
172.	0x29, 0x31, 0xee, 0x90, 0xbc, 0x9d, 0x73, 0xa0,
173.	0x3f, 0xe9, 0x0b, 0xa4, 0xef, 0x30, 0x9b, 0x36,
174.	0x9a, 0xb2, 0x54, 0x77, 0x81, 0x07, 0x4b, 0xaa,
175.	0xa5, 0x77, 0x98, 0xe1, 0xeb, 0xb5, 0x7c, 0x4e,
176.	0x48, 0xd5, 0x08, 0xfc, 0x2c, 0x36, 0xe2, 0x65,
177.	0x03, 0xac, 0xe5, 0xf3, 0x96, 0xb7, 0xd0, 0xb5,
178.	0x3b, 0x92, 0xe4, 0x14, 0x05, 0x7a, 0x6a, 0x92,
179.	0x56, 0xfe, 0x4e, 0xab, 0xd3, 0x0e, 0x32, 0x04,
180.	0x22, 0x22, 0x74, 0x47, 0x7d, 0xec, 0x21, 0x99,
181.	0x30, 0x31, 0x64, 0x46, 0x64, 0x9b, 0xc7, 0x13,
182.	0xbf, 0xbe, 0xd0, 0x31, 0x49, 0xe7, 0x3c, 0xbf,
183.	0xba, 0xb1, 0x20, 0xf9, 0x42, 0xf4, 0xa9, 0xa9,
184.	0xe5, 0x13, 0x65, 0x32, 0xbf, 0x7c, 0xcc, 0x91,
185.	0xd3, 0xfd, 0x24, 0x47, 0x0b, 0xe5, 0x53, 0xad,
186.	0x50, 0x30, 0x56, 0xd1, 0xfa, 0x9c, 0x37, 0xa8,
187.	0xc1, 0xce, 0xf6, 0x0b, 0x18, 0xaa, 0x7c, 0xab,
188.	0xbd, 0x1f, 0xdf, 0xe4, 0x80, 0xb8, 0xa7, 0xe0,
189.	0xad, 0x7d, 0x50, 0x74, 0xf1, 0x98, 0x78, 0xbc,
190.	0x58, 0xb9, 0xc2, 0x52, 0xbe, 0xd2, 0x5b, 0x81,
191.	0x94, 0x83, 0x8f, 0xb9, 0x4c, 0xee, 0x01, 0x2b,
192.	0x5e, 0xc9, 0x6e, 0x9b, 0xf5, 0x63, 0x69, 0xe4,
193.	0xd8, 0x0b, 0x47, 0xd8, 0xfd, 0xd8, 0xe0, 0xed,
194.	0xa8, 0x27, 0x03, 0x74, 0x1e, 0x5d, 0x32, 0xe6,
195.	0x5c, 0x63, 0xc2, 0xfb, 0x3f, 0xee, 0xb4, 0x13,

196.	0xc6, 0x0e, 0x6e, 0x74, 0xe0, 0x22, 0xac, 0xce,
197.	0x79, 0xf9, 0x43, 0x68, 0xc1, 0x03, 0x74, 0x2b,
198.	0xe1, 0x18, 0xf8, 0x7f, 0x76, 0x9a, 0xea, 0x82,
199.	0x3f, 0xc2, 0xa6, 0xa7, 0x4c, 0xf e, 0xae, 0x29,
200.	0x3b, 0xc1, 0x10, 0x7c, 0xd5, 0x77, 0x17, 0x79,
201.	0x5f, 0xcb, 0xad, 0x1f, 0xd8, 0xa1, 0xf d, 0x90,
202.	0xe1, 0x6b, 0xb2, 0xef, 0xb9, 0x41, 0x26, 0xa4,
203.	0x0b, 0x4f, 0xc6, 0x83, 0x05, 0x6f, 0xf0, 0x64,
204.	0x40, 0xe1, 0x44, 0xc4, 0xf9, 0x40, 0x2b, 0x3b,
205.	0x40, 0xdb, 0xaf, 0x35, 0xa4, 0x9b, 0x9f, 0xc4,
206.	0x74, 0x07, 0xe5, 0x18, 0x60, 0xc5, 0xf e, 0x15,
207.	0x0e, 0x3a, 0x25, 0x2a, 0x11, 0xee, 0x78, 0x2f,
208.	0xb8, 0xd1, 0x6e, 0x4e, 0x3c, 0x0a, 0xb5, 0xb9,
209.	0x40, 0x86, 0x27, 0x6d, 0x8f, 0x53, 0xb7, 0x77,
210.	0x36, 0xec, 0x5d, 0xed, 0x32, 0x40, 0x43, 0x82,
211.	0xc3, 0x52, 0x58, 0xc4, 0x26, 0x39, 0xf3, 0xb3,
212.	0xad, 0x58, 0xab, 0xb7, 0xf7, 0x8e, 0x0e, 0xba,
213.	0x8e, 0x78, 0x9d, 0xbf, 0x58, 0x34, 0xbd, 0x77,
214.	0x73, 0xa6, 0x50, 0x55, 0x00, 0x60, 0x26, 0xbf,
215.	0x6d, 0xb4, 0x98, 0x8a, 0x18, 0x83, 0x89, 0xf8,
216.	0xcd, 0x0d, 0x49, 0x06, 0xae, 0x51, 0x6e, 0xaf,
217.	0xbd, 0xe2, 0x07, 0x13, 0xd8, 0x64, 0xcc, 0xbf,
218.	0x00, 0x00, 0x00, 0x01, 0x00, 0x05, 0x58, 0x2e,
219.	0x35, 0x30, 0x39, 0x00, 0x00, 0x02, 0x34, 0x30,
220.	0x82, 0x02, 0x30, 0x30, 0x82, 0x01, 0xda, 0xa0,
221.	0x03, 0x02, 0x01, 0x02, 0x02, 0x04, 0x48, 0x59,
222.	0xf2, 0x84, 0x30, 0x0d, 0x06, 0x09, 0x2a, 0x86,
223.	0x48, 0x86, 0xf7, 0x0d, 0x01, 0x01, 0x05, 0x05,
224.	0x00, 0x30, 0x81, 0x9d, 0x31, 0x0b, 0x30, 0x09,
225.	0x06, 0x03, 0x55, 0x04, 0x06, 0x13, 0x02, 0x4b,
226.	0x52, 0x31, 0x13, 0x30, 0x11, 0x06, 0x03, 0x55,
227.	0x04, 0x08, 0x13, 0x0a, 0x4b, 0x79, 0x75, 0x6e,
228.	0x67, 0x67, 0x69, 0x2d, 0x64, 0x6f, 0x31, 0x14,
229.	0x30, 0x12, 0x06, 0x03, 0x55, 0x04, 0x07, 0x13,
230.	0x0b, 0x53, 0x65, 0x6f, 0x6e, 0x67, 0x6e, 0x61,
231.	0x6d, 0x2d, 0x73, 0x69, 0x31, 0x1a, 0x30, 0x18,
232.	0x06, 0x03, 0x55, 0x04, 0x0a, 0x13, 0x11, 0x54,
233.	0x68, 0x65, 0x20, 0x4e, 0x65, 0x74, 0x74, 0x79,
234.	0x20, 0x50, 0x72, 0x6f, 0x6a, 0x65, 0x63, 0x74,
235.	0x31, 0x15, 0x30, 0x13, 0x06, 0x03, 0x55, 0x04,
236.	0x0b, 0x13, 0x0c, 0x43, 0x6f, 0x6e, 0x74, 0x72,
237.	0x69, 0x62, 0x75, 0x74, 0x6f, 0x72, 0x73, 0x31,
238.	0x30, 0x30, 0x2e, 0x06, 0x03, 0x55, 0x04, 0x03,
239.	0x13, 0x27, 0x73, 0x65, 0x63, 0x75, 0x72, 0x65,
240.	0x63, 0x68, 0x61, 0x74, 0x2e, 0x65, 0x78, 0x61,
241.	0x6d, 0x70, 0x6c, 0x65, 0x2e, 0x6e, 0x65, 0x74,
242.	0x74, 0x79, 0x2e, 0x67, 0x6c, 0x65, 0x61, 0x6d,
243.	0x79, 0x6e, 0x6f, 0x64, 0x65, 0x2e, 0x6e, 0x65,
244.	0x74, 0x30, 0x20, 0x17, 0x0d, 0x30, 0x38, 0x30,
245.	0x36, 0x31, 0x39, 0x30, 0x35, 0x34, 0x35, 0x34,
246.	0x30, 0x5a, 0x18, 0x0f, 0x32, 0x31, 0x38, 0x37,
247.	0x31, 0x31, 0x32, 0x33, 0x30, 0x35, 0x34, 0x35,
248.	0x34, 0x30, 0x5a, 0x30, 0x81, 0x9d, 0x31, 0x0b,
249.	0x30, 0x09, 0x06, 0x03, 0x55, 0x04, 0x06, 0x13,
250.	0x02, 0x4b, 0x52, 0x31, 0x13, 0x30, 0x11, 0x06,
251.	0x03, 0x55, 0x04, 0x08, 0x13, 0x0a, 0x4b, 0x79,
252.	0x75, 0x6e, 0x67, 0x67, 0x69, 0x2d, 0x64, 0x6f,
253.	0x31, 0x14, 0x30, 0x12, 0x06, 0x03, 0x55, 0x04,
254.	0x07, 0x13, 0x0b, 0x53, 0x65, 0x6f, 0x6e, 0x67,
255.	0x6e, 0x61, 0x6d, 0x2d, 0x73, 0x69, 0x31, 0x1a,
256.	0x30, 0x18, 0x06, 0x03, 0x55, 0x04, 0x0a, 0x13,
257.	0x11, 0x54, 0x68, 0x65, 0x20, 0x4e, 0x65, 0x74,
258.	0x74, 0x79, 0x20, 0x50, 0x72, 0x6f, 0x6a, 0x65,
259.	0x63, 0x74, 0x31, 0x15, 0x30, 0x13, 0x06, 0x03,
260.	0x55, 0x04, 0x0b, 0x13, 0x0c, 0x43, 0x6f, 0x6e,
261.	0x74, 0x72, 0x69, 0x62, 0x75, 0x74, 0x6f, 0x72,
262.	0x73, 0x31, 0x30, 0x30, 0x2e, 0x06, 0x03, 0x55,
263.	0x04, 0x03, 0x13, 0x27, 0x73, 0x65, 0x63, 0x75,
264.	0x72, 0x65, 0x63, 0x68, 0x61, 0x74, 0x2e, 0x65,
265.	0x78, 0x61, 0x6d, 0x70, 0x6c, 0x65, 0x2e, 0x6e,
266.	0x65, 0x74, 0x74, 0x79, 0x2e, 0x67, 0x6c, 0x65,
267.	0x61, 0x6d, 0x79, 0x6e, 0x6f, 0x64, 0x65, 0x2e,
268.	0x6e, 0x65, 0x74, 0x30, 0x5c, 0x30, 0x0d, 0x06,
269.	0x09, 0x2a, 0x86, 0x48, 0x86, 0xf7, 0x0d, 0x01,
270.	0x01, 0x01, 0x05, 0x00, 0x03, 0x4b, 0x00, 0x30,
271.	0x48, 0x02, 0x41, 0x00, 0x95, 0xb3, 0x47, 0x17,
272.	0x95, 0x0f, 0x57, 0xcf, 0x66, 0x72, 0x0a, 0x7e,
273.	0x5b, 0x54, 0xea, 0x8c, 0x6f, 0x79, 0xde, 0x94,
274.	0xac, 0x0b, 0x5a, 0xd4, 0xd6, 0x1b, 0x58, 0x12,

```
275.         0x1a, 0x16, 0x3d, 0xfe, 0xdf, 0xa5, 0x2b, 0x86,
276.         0xbc, 0x64, 0xd4, 0x80, 0x1e, 0x3f, 0xf9, 0xe2,
277.         0x04, 0x03, 0x79, 0x9b, 0xc1, 0x5c, 0xf0, 0xf1,
278.         0xf3, 0xf1, 0xe3, 0xbf, 0x3f, 0xc0, 0x1f, 0xdd,
279.         0xdb, 0xc0, 0x5b, 0x21, 0x02, 0x03, 0x01, 0x00,
280.         0x01, 0x30, 0x0d, 0x06, 0x09, 0x2a, 0x86, 0x48,
281.         0x86, 0xf7, 0x0d, 0x01, 0x01, 0x05, 0x05, 0x00,
282.         0x03, 0x41, 0x00, 0x02, 0xd7, 0xdd, 0xbd, 0x0c,
283.         0x8e, 0x21, 0x20, 0xef, 0x9e, 0x4f, 0x1f, 0xf5,
284.         0x49, 0xf1, 0xae, 0x58, 0x9b, 0x94, 0x3a, 0x1f,
285.         0x70, 0x33, 0xf0, 0x9b, 0xbb, 0xe9, 0xc0, 0xf3,
286.         0x72, 0xcb, 0xde, 0xb6, 0x56, 0x72, 0xcc, 0x1c,
287.         0xf0, 0xd6, 0x5a, 0x2a, 0xbc, 0xa1, 0x7e, 0x23,
288.         0x83, 0xe9, 0xe7, 0xcf, 0x9e, 0xa5, 0xf9, 0xcc,
289.         0xc2, 0x61, 0xf4, 0xdb, 0x40, 0x93, 0x1d, 0x63,
290.         0x8a, 0x50, 0x4c, 0x11, 0x39, 0xb1, 0x91, 0xc1,
291.         0xe6, 0x9d, 0xd9, 0x1a, 0x62, 0x1b, 0xb8, 0xd3,
292.         0xd6, 0x9a, 0x6d, 0xb9, 0x8e, 0x15, 0x51 };
293.
294.     public static InputStream asInputStream() {
295.         byte[] data = new byte[DATA.length];
296.         for (int i = 0; i < data.length; i++) {
297.             data[i] = (byte) DATA[i];
298.         }
299.         return new ByteArrayInputStream(data);
300.     }
301.
302.     public static char[] getCertificatePassword() {
303.         return "secret".toCharArray();
304.     }
305.
306.     public static char[] getKeyStorePassword() {
307.         return "secret".toCharArray();
308.     }
309.
310.     private SecureChat KeyStore() {
311.         // Unused
312.     }
313. }
```

```
[java]
01.  /*
02.   * Copyright 2012 The Netty Project
03.   *
04.   * The Netty Project licenses this file to you under the Apache License,
05.   * version 2.0 (the "License"); you may not use this file except in compliance
06.   * with the License. You may obtain a copy of the License at:
07.   *
08.   * http://www.apache.org/licenses/LICENSE-2.0
09.   *
10.   * Unless required by applicable law or agreed to in writing, software
11.   * distributed under the License is distributed on an "AS IS" BASIS, WITHOUT
12.   * WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
13.   * License for the specific language governing permissions and limitations
14.   * under the License.
15.   */
16. package netty.in.action.spdy;
17.
18. import io.netty.handler.ssl.SslHandler;
19. import io.netty.util.internal.SystemPropertyUtil;
20.
21. import java.security.KeyStore;
22. import java.security.SecureRandom;
23.
24. import javax.net.ssl.KeyManager;
25. import javax.net.ssl.KeyManagerFactory;
26. import javax.net.ssl.SSLContext;
27. import javax.net.ssl.SSLEngine;
28. import javax.net.ssl.TrustManager;
29.
30. /**
31.  * Creates a bogus {@link SSLContext}. A client-side context created by this
32.  * factory accepts any certificate even if it is invalid. A server-side context
33.  * created by this factory sends a bogus certificate defined in {@link SecureChat KeyStore}.
34.  * <p>
35.  * You will have to create your context differently in a real world application.
36.  *
37.  * <h3>Client Certificate Authentication</h3>
```

```
38. *
39. * To enable client certificate authentication:
40. * <ul>
41. * <li>Enable client authentication on the server side by calling
42. *     {@link SSLEngine#setNeedClientAuth(boolean)} before creating
43. *     {@link SslHandler}.</li>
44. * <li>When initializing an {@link SSLContext} on the client side,
45. *     specify the {@link KeyManager} that contains the client certificate as
46. *     the first argument of {@link SSLContext#init(KeyManager[], TrustManager[], SecureRandom)}.</li>
47. * <li>When initializing an {@link SSLContext} on the server side,
48. *     specify the proper {@link TrustManager} as the second argument of
49. *     {@link SSLContext#init(KeyManager[], TrustManager[], SecureRandom)}
50. *     to validate the client certificate.</li>
51. * </ul>
52. */
53. public final class SecureChatSslContextFactory {
54.
55.     private static final String PROTOCOL = "TLS";
56.     private static final SSLContext SERVER_CONTEXT;
57.     private static final SSLContext CLIENT_CONTEXT;
58.
59.     static {
60.         String algorithm = SystemPropertyUtil.get("ssl.KeyManagerFactory.algorithm");
61.         if (algorithm == null) {
62.             algorithm = "SunX509";
63.         }
64.
65.         SSLContext serverContext;
66.         SSLContext clientContext;
67.         try {
68.             KeyStore ks = KeyStore.getInstance("JKS");
69.             ks.load(SecureChatKeyStore.asInputStream(),
70.                 SecureChatKeyStore.getKeyStorePassword());
71.
72.             // Set up key manager factory to use our key store
73.             KeyManagerFactory knf = KeyManagerFactory.getInstance(algorithm);
74.             knf.init(ks, SecureChatKeyStore.getCertificatePassword());
75.
76.             // Initialize the SSLContext to work with our key managers.
77.             serverContext = SSLContext.getInstance(PROTOCOL);
78.             serverContext.init(knf.getKeyManagers(), null, null);
79.         } catch (Exception e) {
80.             throw new Error(
81.                 "Failed to initialize the server-side SSLContext", e);
82.         }
83.
84.         try {
85.             clientContext = SSLContext.getInstance(PROTOCOL);
86.             clientContext.init(null, SecureChatTrustManagerFactory.getTrustManagers(), null);
87.         } catch (Exception e) {
88.             throw new Error(
89.                 "Failed to initialize the client-side SSLContext", e);
90.         }
91.
92.         SERVER_CONTEXT = serverContext;
93.         CLIENT_CONTEXT = clientContext;
94.     }
95.
96.     public static SSLContext getServerContext() {
97.         return SERVER_CONTEXT;
98.     }
99.
100.    public static SSLContext getClientContext() {
101.        return CLIENT_CONTEXT;
102.    }
103.
104.    private SecureChatSslContextFactory() {
105.        // Unused
106.    }
107. }
```

12.4 Summary

这一章没有详细的按照netty in action书中来翻译，因为我感觉书中讲的很多都不是netty的重点，鄙人英文能实在有限，所以也就把精力不放在非核心上面了。若有读者需要详细在netty中使用spdy可以查看其它相关资料或文章，或者看本篇博文的例子代码。后面几章也会如此。

第十三章：通过UDP广播事件

目录

[1]

1. UDP介绍

2. UDP程序结构和设计

3. 日志事件POJO

4. 编写广播器

5. 编写监听者

6. 使用LogEventBroadcaster和LogEventMonitor

7. Summary

本章介绍

- UDP介绍
- UDP程序结构和设计
- 日志事件POJO
- 编写广播器
- 编写监听者
- 使用广播器和监听者
- Summary

前面的章节都是在示例中使用TCP协议，这一章，我们将使用UDP。UDP是一种无连接协议，若需要很高的性能和对数据的完成性没有严格要求，那使用UDP是一个很好的方法。最著名的基于UDP协议的是用来域名解析的DNS。

Netty使用了统一的传输API，这使得编写基于UDP的应用程序很容易。可以重用现有的ChannelHandler和其他公共组件来编写另外的Netty程序。看完本章后，你就会知道什么事无连接协议以及为什么UDP可能适合你的应用程序。

13.1 UDP介绍

在深入探讨UDP之前，我们先了解UDP是什么，以及UDP有什么限制或问题。UDP是一种无连接的协议，也就是说客户端和服务端在交互数据之前不会像TCP那样事先建立连接。

UDP是User Datagram Protocol的简称，即用户数据报协议。UDP有不提供数据报分组、组装和不能对数据报进行排序的缺点，也就是说，当数据报发送之后是无法确认数据是否完整到达的。

UDP协议的主要作用是将网络数据流量压缩成数据包的形式。一个典型的数据包就是一个二进制数据的传输单位。每一个数据包的前8个字节用来包含报头信息，剩余字节则用来包含具体的传输数据。

在选择使用协议的时候，选择UDP必须要谨慎。在网络质量令人十分不满意的坏境下，UDP协议数据包丢失会比较严重。但是由于UDP的特性：它不属于连接型协议，因而具有资源消耗小，处理速度快的优点，所以通常音频、视频和普通数据在传送时使用UDP较多，因为它们即使偶尔丢失一两个数据包，也不会对接收结果产生太大影响。比如我们聊天用的ICQ和QQ就是使用的UDP协议。

UDP就介绍到这里，更详细的资料可以百度或谷歌。

13.2 UDP程序结构和设计

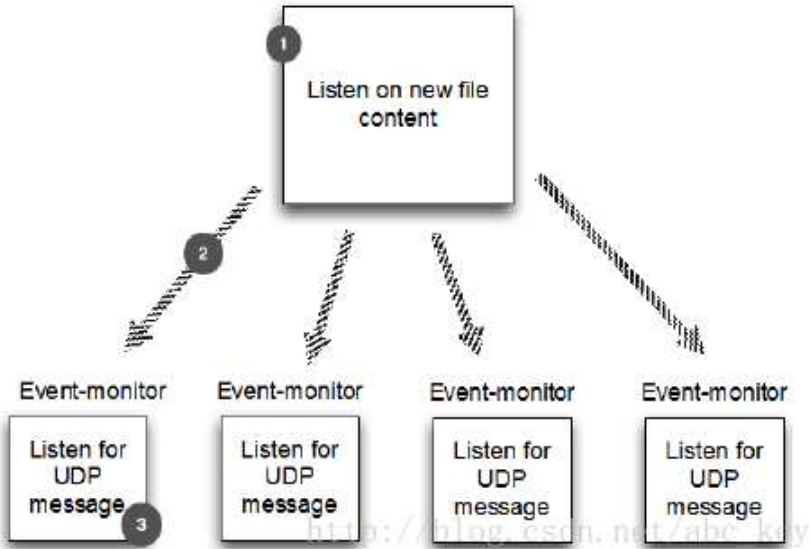
本章例子中，程序打开一个文件并将文件内容一行一行的通过UDP广播到其他的接收主机，这很像UNIX操作系统的日志系统。对于像发送日志的需求，UDP非常适合这样的应用程序，并可以使用UDP通过网络发送大量的“事件”。

使用UDP可以在同一个主机上启动多个应用程序并能独立的进行数据报的发送和接收，UDP使用底层的互联网协议来传送报文，同IP一样提供不可靠的无连接数据报传输服务，它不提供报文到达确认、排序、及流量控制等功能。每个UDP报文分UDP报头和UDP数据区两部分，报头由四个16位长（2字节）字段组成，分别说明该报文的源端口、目的端口、报文长度以及校验值；数据库就是传输的具体数据。

UDP最好在局域网内使用，这样可以大大减少丢包概率。UDP有如下特性：

1. UDP是一个无连接协议，传输数据之前源端和终端不建立连接，当它想传送时就简单地去抓取来自应用程序的数据，并尽可能快地把它扔到网络上。在发送端，UDP传送数据的速度仅仅是受应用程序生成数据的速度、计算机的能力和传输带宽的限制；在接收端，UDP把每个消息段放在队列中，应用程序每次从队列中读一个消息段。
2. 由于传输数据不建立连接，因此也就不需要维护连接状态，包括收发状态等，因此一台服务器可同时向多个客户机传输相同的消息。
3. UDP信息包的标题很短，只有8个字节，相对于TCP的20个字节信息包的额外开销很小。
4. 吞吐量不受拥挤控制算法的调节，只受应用软件生成数据的速率、传输带宽、源端和终端主机性能的限制。
5. UDP使用尽最大努力交付，即不保证可靠交付，因此主机不需要维持复杂的链接状态表（这里面有许多参数）。
6. UDP是面向报文的。发送方的UDP对应用程序交下来的报文，在添加首部后就向下交付给IP层。既不拆分，也不合并，而是保留这些报文的边界，因此，应用程序需要选择合适的报文大小。

本章UDP程序例子的示意图入如下：



从上图可以看出，例子程序由两部分组成：广播日志文件和“监控器”，监控器用于接收广播。为了简单，我们将不做任何形式的身份验证或加密。

13.3 日志事件POJO

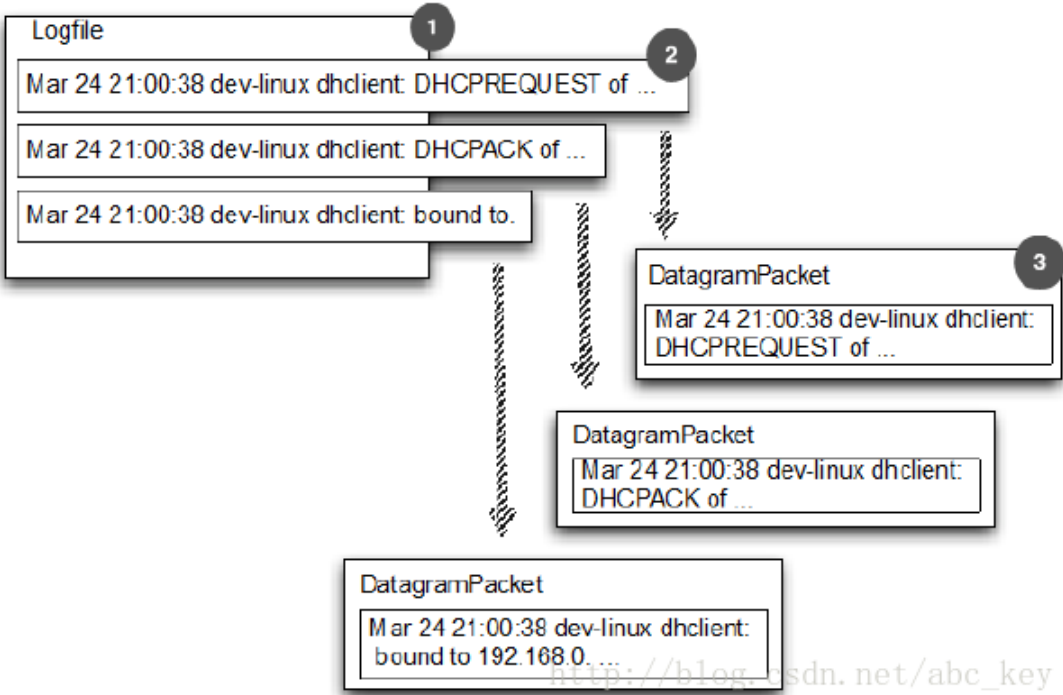
我们的应用程序通常需要某种“消息POJO”用于保存消息，我们把这个消息POJO看成是一个“事件消息”在本例子中我们也创建一个POJO叫做LogEvent，LogEvent用来存储事件数据，然后将数据输出到日志文件。看下面代码：

```
[java]
01. package netty.in.action.udp;
02.
03. import java.net.InetSocketAddress;
04.
05. public class LogEvent {
06.
07.     public static final byte SEPARATOR = (byte) '|';
08.
09.     private final InetSocketAddress source;
10.     private final String logfile;
11.     private final String msg;
12.     private final long received;
13.
14.     public LogEvent(String logfile, String msg) {
15.         this(null, -1, logfile, msg);
16.     }
17.
18.     public LogEvent(InetSocketAddress source, long received, String logfile, String msg) {
19.         this.source = source;
20.         this.logfile = logfile;
21.         this.msg = msg;
22.         this.received = received;
23.     }
24.
25.     public InetSocketAddress getSource() {
26.         return source;
27.     }
28.
29.     public String getLogfile() {
30.         return logfile;
31.     }
32.
33.     public String getMsg() {
34.         return msg;
35.     }
36.
37.     public long getReceived() {
38.         return received;
39.     }
40.
41. }
```

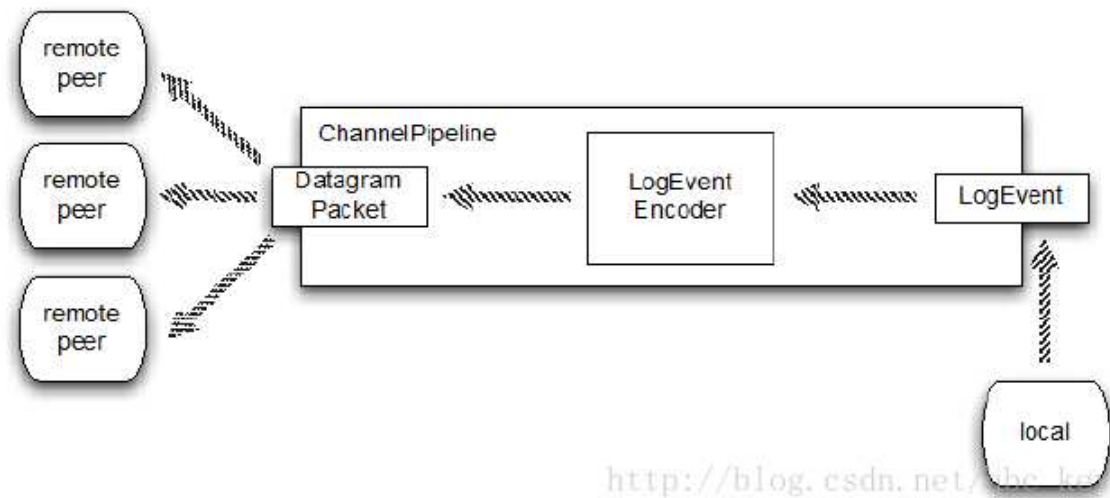
接下来的章节，我们将用这个POJO类来实现具体的逻辑。

13.4 编写广播器

我们要做的是广播一个DatagramPacket日志条目，如下图所示：



上图显示我们有一个从日志条路到DatagramPacket一对一的关系。如同所有的基于Netty的应用程序一样，它由一个或多个ChannelHandler和一些实体对象绑定，用于引导该应用程序。首先让我们来看看LogEventBroadcaster的ChannelPipeline以及作为数据载体的LogEvent的流向，看下图：



上图显示，LogEventBroadcaster使用LogEvent消息并将消息写入本地Channel，所有的信息封装在LogEvent消息中，这些消息被传到ChannelPipeline中。流进ChannelPipeline的LogEvent消息被编码成DatagramPacket消息，最后通过UDP广播到远程对等通道。

这可以归结为有一个自定义的ChannelHandler，从LogEvent消息编程成DatagramPacket消息。回忆我们在第七章讲解的编解码器，我们定义个LogEventEncoder，代码如下：

```
[ java ]
01. package netty.in.action.udp;
02.
03. import io.netty.buffer.ByteBuf;
04. import io.netty.channel.ChannelHandlerContext;
05. import io.netty.channel.socket.DatagramPacket;
06. import io.netty.handler.codec.MessageToMessageEncoder;
07. import io.netty.util.CharsetUtil;
08.
09. import java.net.InetSocketAddress;
10. import java.util.List;
11.
12. public class LogEventEncoder extends MessageToMessageEncoder<LogEvent> {
13.
14.     private final InetSocketAddress remoteAddress;
15.
16.     public LogEventEncoder(InetSocketAddress remoteAddress) {
17.         this.remoteAddress = remoteAddress;
18.     }
19.
20.     @Override
21.     protected void encode(ChannelHandlerContext ctx, LogEvent msg, List<Object> out)
22.         throws Exception {
23.         ByteBuf buf = ctx.alloc().buffer();
24.         buf.writeBytes(msg.getLogfile().getBytes(CharsetUtil.UTF_8));
25.         buf.writeByte(LogEvent.SEPARATOR);
26.         buf.writeBytes(msg.getMg().getBytes(CharsetUtil.UTF_8));
27.         out.add(new DatagramPacket(buf, remoteAddress));
28.     }
```

```
29.
30. }
```

下面我们再编写一个广播器：

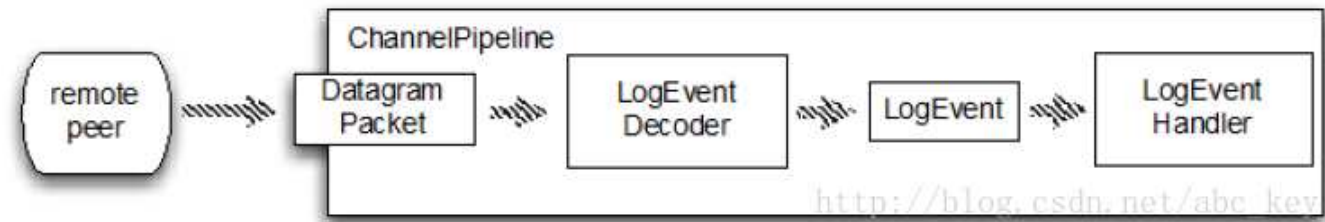
```
[java]
01. package netty.in.action.udp;
02.
03. import io.netty.bootstrap.Bootstrap;
04. import io.netty.channel.Channel;
05. import io.netty.channel.ChannelOption;
06. import io.netty.channel.EventLoopGroup;
07. import io.netty.channel.nio.NioEventLoopGroup;
08. import io.netty.channel.socket.nio.NioDatagramChannel;
09.
10. import java.io.File;
11. import java.io.IOException;
12. import java.io.RandomAccessFile;
13. import java.net.InetSocketAddress;
14.
15. public class LogEventBroadcaster {
16.
17.     private final EventLoopGroup group;
18.     private final Bootstrap bootstrap;
19.     private final File file;
20.
21.     public LogEventBroadcaster(InetSocketAddress address, File file) {
22.         group = new NioEventLoopGroup();
23.         bootstrap = new Bootstrap();
24.         bootstrap.group(group).channel(NioDatagramChannel.class)
25.             .option(ChannelOption.SO_BROADCAST, true)
26.             .handler(new LogEventEncoder(address));
27.         this.file = file;
28.     }
29.
30.     public void run() throws IOException {
31.         Channel ch = bootstrap.bind(0).syncUninterruptibly().channel();
32.         long pointer = 0;
33.         for (;;) {
34.             long len = file.length();
35.             if (len < pointer) {
36.                 pointer = len;
37.             } else {
38.                 RandomAccessFile raf = new RandomAccessFile(file, "r");
39.                 raf.seek(pointer);
40.                 String line;
41.                 while ((line = raf.readLine()) != null) {
42.                     ch.write(new LogEvent(null, -1, file.getAbsolutePath(), line));
43.                 }
44.                 ch.flush();
45.                 pointer = raf.getFilePointer();
46.                 raf.close();
47.             }
48.             try {
49.                 Thread.sleep(1000);
50.             } catch (InterruptedException e) {
51.                 Thread.interrupted();
52.                 break;
53.             }
54.         }
55.     }
56.
57.     public void stop() {
58.         group.shutdownGracefully();
59.     }
60.
61.     public static void main(String[] args) throws Exception {
62.         int port = 4096;
63.         String path = System.getProperty("user.dir") + "/log.txt";
64.         LogEventBroadcaster broadcaster = new LogEventBroadcaster(new InetSocketAddress(
65.             "255.255.255.255", port), new File(path));
66.         try {
67.             broadcaster.run();
68.         } finally {
69.             broadcaster.stop();
70.         }
71.     }
72. }
```

13.5 编写监听者

这一节我们编写一个监听者：EventLogMonitor，也就是用来接收数据的程序。EventLogMonitor做下面事情：

- 接收LogEventBroadcaster广播的DatagramPacket
- 解码LogEvent消息
- 输出LogEvent

EventLogMonitor的示意图如下：



解码器代码如下：

```
[java]

01. package netty.in.action.udp;
02.
03. import io.netty.buffer.ByteBuf;
04. import io.netty.channel.ChannelHandlerContext;
05. import io.netty.channel.socket.DatagramPacket;
06. import io.netty.handler.codec.MessageToMessageDecoder;
07. import io.netty.util.CharsetUtil;
08.
09. import java.util.List;
10.
11. public class LogEventDecoder extends MessageToMessageDecoder<DatagramPacket> {
12.
13.     @Override
14.     protected void decode(ChannelHandlerContext ctx, DatagramPacket msg, List<Object> out)
15.         throws Exception {
16.         ByteBuf buf = msg.content();
17.         int i = buf.indexOf(0, buf.readableBytes(), LogEvent.SEPARATOR);
18.         String filename = buf.slice(0, i).toString(CharsetUtil.UTF_8);
19.         String logMsg = buf.slice(i + 1, buf.readableBytes()).toString(CharsetUtil.UTF_8);
20.         LogEvent event = new LogEvent(msg.sender(),
21.             System.currentTimeMillis(), filename, logMsg);
22.         out.add(event);
23.     }
24.
25. }
```

处理消息的Handler代码如下：

```
[java]

01. package netty.in.action.udp;
02.
03. import io.netty.channel.ChannelHandlerContext;
04. import io.netty.channel.SimpleChannelInboundHandler;
05.
06. public class LogEventHandler extends SimpleChannelInboundHandler<LogEvent> {
07.
08.     @Override
09.     protected void channelRead0(ChannelHandlerContext ctx, LogEvent msg) throws Exception {
10.         StringBuilder builder = new StringBuilder();
11.         builder.append(msg.getReceived());
12.         builder.append(" [");
13.         builder.append(msg.getSource().toString());
14.         builder.append("] [");
15.         builder.append(msg.getLogfile());
16.         builder.append("] : ");
17.         builder.append(msg.getMg());
18.         System.out.println(builder.toString());
19.     }
20. }
```

EventLogMonitor代码如下：

```
[java]
```



```
01. package netty.in.action.udp;
02.
03. import io.netty.bootstrap.Bootstrap;
04. import io.netty.channel.Channel;
05. import io.netty.channel.ChannelInitializer;
06. import io.netty.channel.ChannelOption;
07. import io.netty.channel.ChannelPipeline;
08. import io.netty.channel.EventLoopGroup;
09. import io.netty.channel.nio.NioEventLoopGroup;
10. import io.netty.channel.socket.nio.NioDatagramChannel;
11.
12. import java.net.InetSocketAddress;
13.
14. public class LogEventMonitor {
15.
16.     private final EventLoopGroup group;
17.     private final Bootstrap bootstrap;
18.
19.     public LogEventMonitor(InetSocketAddress address) {
20.         group = new NioEventLoopGroup();
21.         bootstrap = new Bootstrap();
22.         bootstrap.group(group).channel(NioDatagramChannel.class)
23.             .option(ChannelOption.SO_BROADCAST, true)
24.             .handler(new ChannelInitializer<Channel>() {
25.                 @Override
26.                 protected void initChannel(Channel channel) throws Exception {
27.                     ChannelPipeline pipeline = channel.pipeline();
28.                     pipeline.addLast(new LogEventDecoder());
29.                     pipeline.addLast(new LogEventHandler());
30.                 }
31.             }).localAddress(address);
32.     }
33.
34.     public Channel bind() {
35.         return bootstrap.bind().syncUninterruptibly().channel();
36.     }
37.
38.     public void stop() {
39.         group.shutdownGracefully();
40.     }
41.
42.     public static void main(String[] args) throws InterruptedException {
43.
44.         LogEventMonitor monitor = new LogEventMonitor(new InetSocketAddress(4096));
45.         try {
46.             Channel channel = monitor.bind();
47.             System.out.println("LogEventMonitor running");
48.             channel.closeFuture().sync();
49.         } finally {
50.             monitor.stop();
51.         }
52.     }
53. }
```

13.6 使用LogEventBroadcaster和LogEventMonitor

为避免LogEventMonitor接收不到数据，我们必须先启动LogEventMonitor后，再启动LogEventBroadcaster，输出内容这么就不贴图了，读者可以自己运营本例子测试。

13.7 Summary

本章依然没按照原书中的来翻译，主要是以一个例子来说明UDP在Netty中的使用。概念性的东西都是从网上复制的，读者只需要了解UDP的概念再了解清楚例子代码的含义，并试着运行一些例子。