第一章：Netty 介绍

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

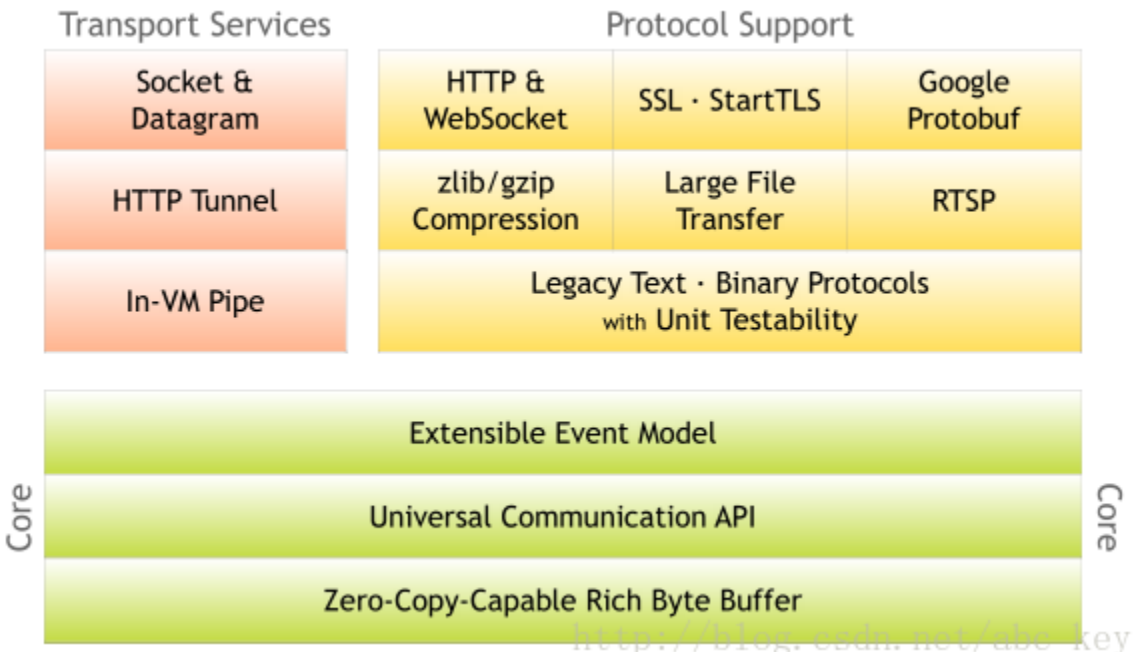
Netty 的“quick and easy(高性能和简单易用)”并不意味着编写的程序的性能和可维护性会受到影响。从 Netty 中实现的协议如 FTP，SMTP，HTTP，WebSocket，SPDY 以及各种二进制和基于文本的传统协议中获得的经验导致 Netty 的创始人要非常小心它的设计。Netty 成功的提供了易于开发，高性能和高稳定性，以及较强的扩展性。

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)	• 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)	• 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+，其特性可能有相关依赖，但都是可选的。
Performance(性能)	• 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)	鲁棒性，可以理解为健壮性 • 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)	• 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)	• 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] view plaincopy

```java
package netty.in.action;

public class Worker {

    public void doWork() {
        Fetcher fetcher = new MyFetcher(new Data(1, 0));
        fetcher.fetchData(new FetcherCallback() {
            @Override
            public void onError(Throwable cause) {
                System.out.println("An error accour: " + cause.getMessage())
;
            }

            @Override
            public void onData(Data data) {
                System.out.println("Data received: " + data);
            }
        });
    }

    public static void main(String[] args) {
        Worker w = new Worker();
        w.doWork();
    }

}
```

[java] view plaincopy

```java
package netty.in.action;

public interface Fetcher {
    void fetchData(FetcherCallback callback);
}
```

[java] view plaincopy

```java
package netty.in.action;
```

```
2.
3.  public class MyFetcher implements Fetcher {
4.
5.      final Data data;
6.
7.      public MyFetcher(Data data){
8.          this.data = data;
9.      }
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.
20. }
```

[java] view plaincopy

```
1.  package netty.in.action;
2.
3.  public interface FetcherCallback {
4.      void onData(Data data) throws Exception;
5.      void onError(Throwable cause);
6.  }
```

[java] view plaincopy

```
1.  package netty.in.action;
2.
3.  public class Data {
4.
5.      private int n;
6.      private int m;
7.
8.      public Data(int n,int m){
9.          this.n = n;
10.         this.m = m;
11.     }
12.
13.     @Override
14.     public String toString() {
```

```
        int r = n/m;
        return n + "/" + m +" = " + r;
    }
}
```

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] view plaincopy

```
package netty.in.action;

import java.util.concurrent.Callable;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class FutureExample {

    public static void main(String[] args) throws Exception {
        ExecutorService executor = Executors.newCachedThreadPool();
        Runnable task1 = new Runnable() {
            @Override
            public void run() {
                //do something
                System.out.println("i am task1.....");
```

```
            }
        };
        Callable<Integer> task2 = new Callable<Integer>() {
            @Override
            public Integer call() throws Exception {
                //do something
                return new Integer(100);
            }
        };
        Future<?> f1 = executor.submit(task1);
        Future<Integer> f2 = executor.submit(task2);
        System.out.println("task1 is completed? " + f1.isDone());
        System.out.println("task2 is completed? " + f2.isDone());
        //waiting task1 completed
        while(f1.isDone()){
            System.out.println("task1 completed.");
            break;
        }
        //waiting task2 completed
        while(f2.isDone()){
            System.out.println("return value by task2: " + f2.get());
            break;
        }
    }

}
```

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 上就会出现问题。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

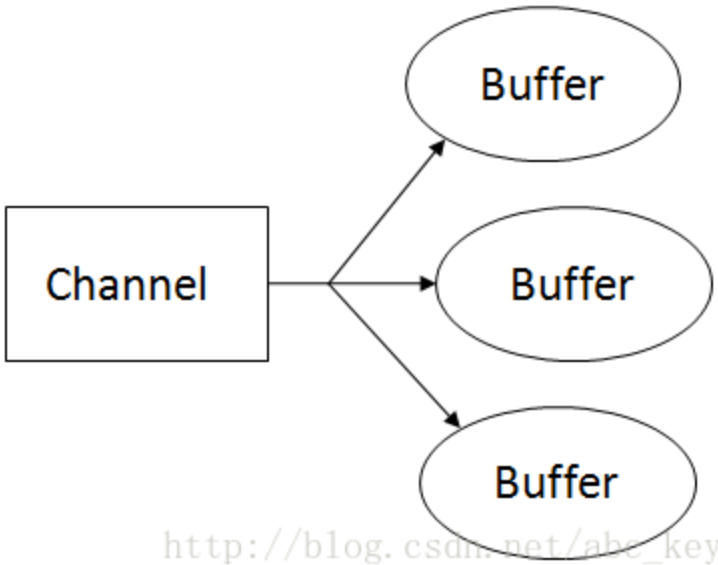
ByteBuffer 是一个数据容器，但是可惜的是 JDK 没有开发 ByteBuffer 实现的源码；ByteBuffer 允许包装一个 byte[]来获得一个实例，如果你希望尽量减少内存拷贝，那么这种方式是非常有用的。若果你想将 ByteBuffer 重新实现，那么不要浪费你的时间了，ByteBuffer 的构造函数是私有的，所以它不能被扩展。Netty 提供了自己的 ByteBuffer 实现，Netty 通过一些简单的 APIs 对 ByteBuffer 进行构造、使用和操作，以此来解决 NIO 中的一些限制。

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

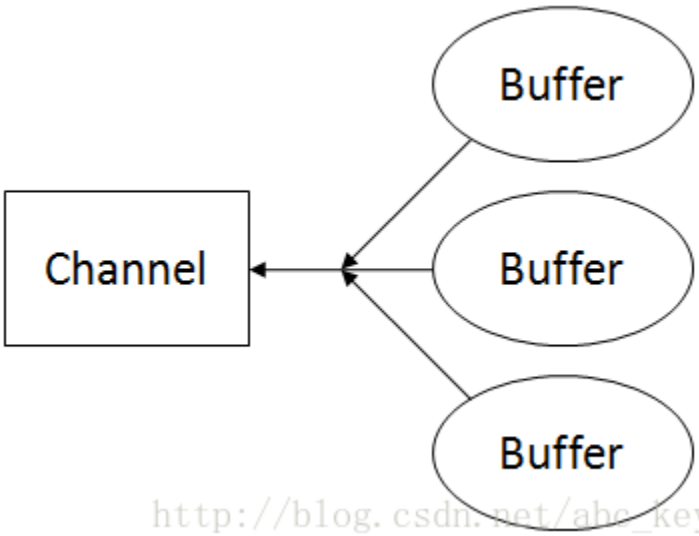
很多 Channel 的实现支持 Gather 和 Scatter。这个功能允许从从多个 ByteBuffer 中读入或写入到过个 ByteBuffer，这样做可以提供性能。操作系统底层知道如何处理这些被写入/读出，并且能以最有效的方式处理。如果要分割的数据再多个不同的 ByteBuffer 中，使用 Gather/Scatter 是比较好的方式。

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.

第二章：第一个 Netty 程序

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：

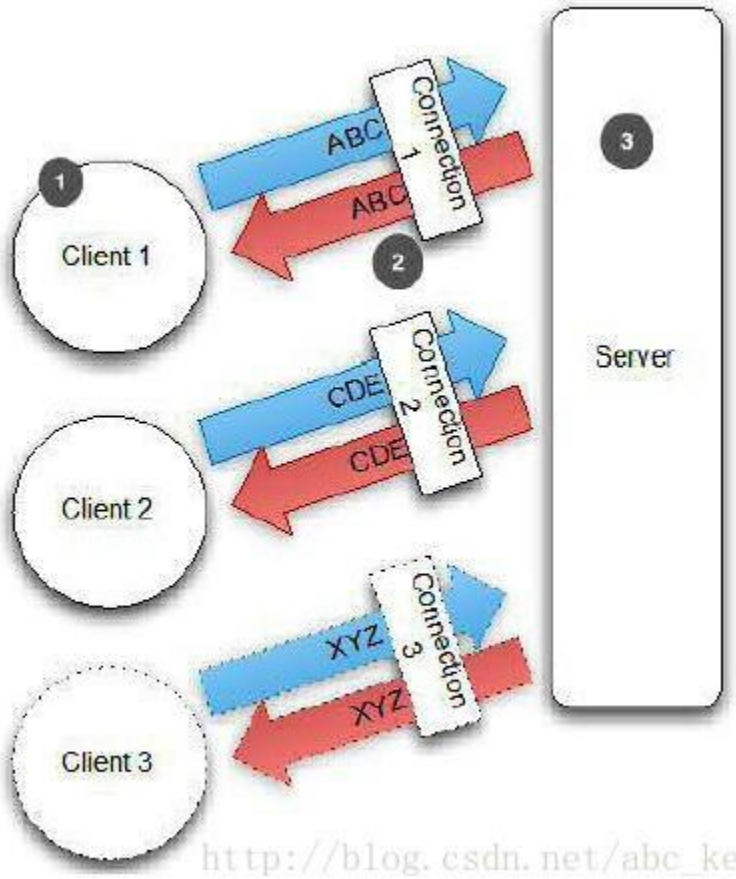
- 安装 JDK7，下载地址 http://www.oracle.com/technetwork/java/javase/archive-139210.html
- 下载 netty 包，下载地址 http://netty.io/
- 安装 Eclipse

《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 客户端和服务器概述

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，但是客户端和服务器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] view plaincopy

```
1. package netty.example;
2. 
3. import io.netty.bootstrap.ServerBootstrap;
4. import io.netty.channel.Channel;
5. import io.netty.channel.ChannelFuture;
6. import io.netty.channel.ChannelInitializer;
7. import io.netty.channel.EventLoopGroup;
8. import io.netty.channel.nio.NioEventLoopGroup;
9. 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).localAddres
    s(port)
27.                     .childHandler(new ChannelInitializer<Channel>() {
28.                         @Override
```

```
29.                         protected void initChannel(Channel ch) throws Except
    ion {
30.                             ch.pipeline().addLast(new EchoServerHandler());

31.                         }
32.                     });
33.             //Binds server, waits for server to close, and releases resource
    s
34.             ChannelFuture f = b.bind().sync();
35.             System.out.println(EchoServer.class.getName() + "started and lis
    ten 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] view plaincopy

```
1.  package netty.example;
2.
3.  import io.netty.buffer.Unpooled;
4.  import io.netty.channel.ChannelFutureListener;
5.  import io.netty.channel.ChannelHandlerContext;
6.  import io.netty.channel.ChannelInboundHandlerAdapter;
7.
8.  public class EchoServerHandler extends ChannelInboundHandlerAdapter {
9.
10.     @Override
11.     public void channelRead(ChannelHandlerContext ctx, Object msg) throws Ex
    ception {
12.         System.out.println("Server received: " + msg);
13.         ctx.write(msg);
14.     }
15.
16.     @Override
17.     public void channelReadComplete(ChannelHandlerContext ctx) throws Except
    ion {
18.         ctx.writeAndFlush(Unpooled.EMPTY_BUFFER).addListener(ChannelFutureLi
    stener.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，比如发生异常后关闭 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] view plaincopy

```
1.  package netty.example;
2.
3.  import io.netty.bootstrap.Bootstrap;
4.  import io.netty.channel.ChannelFuture;
5.  import io.netty.channel.ChannelInitializer;
6.  import io.netty.channel.EventLoopGroup;
7.  import io.netty.channel.nio.NioEventLoopGroup;
8.  import io.netty.channel.socket.SocketChannel;
9.  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] view plaincopy

```
1. package netty.example;
2. 
3. import io.netty.buffer.ByteBuf;
4. import io.netty.buffer.ByteBufUtil;
5. import io.netty.buffer.Unpooled;
6. import io.netty.channel.ChannelHandlerContext;
7. import io.netty.channel.SimpleChannelInboundHandler;
8. import io.netty.util.CharsetUtil;
9. 
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) thro
    ws Exception {
19.         System.out.println("Client received: " + ByteBufUtil.hexDump(msg.rea
    dBytes(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)程序客户端和服务器

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。

第三章：Netty 核心概念

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(客户端和服务器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 中来处理的，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。

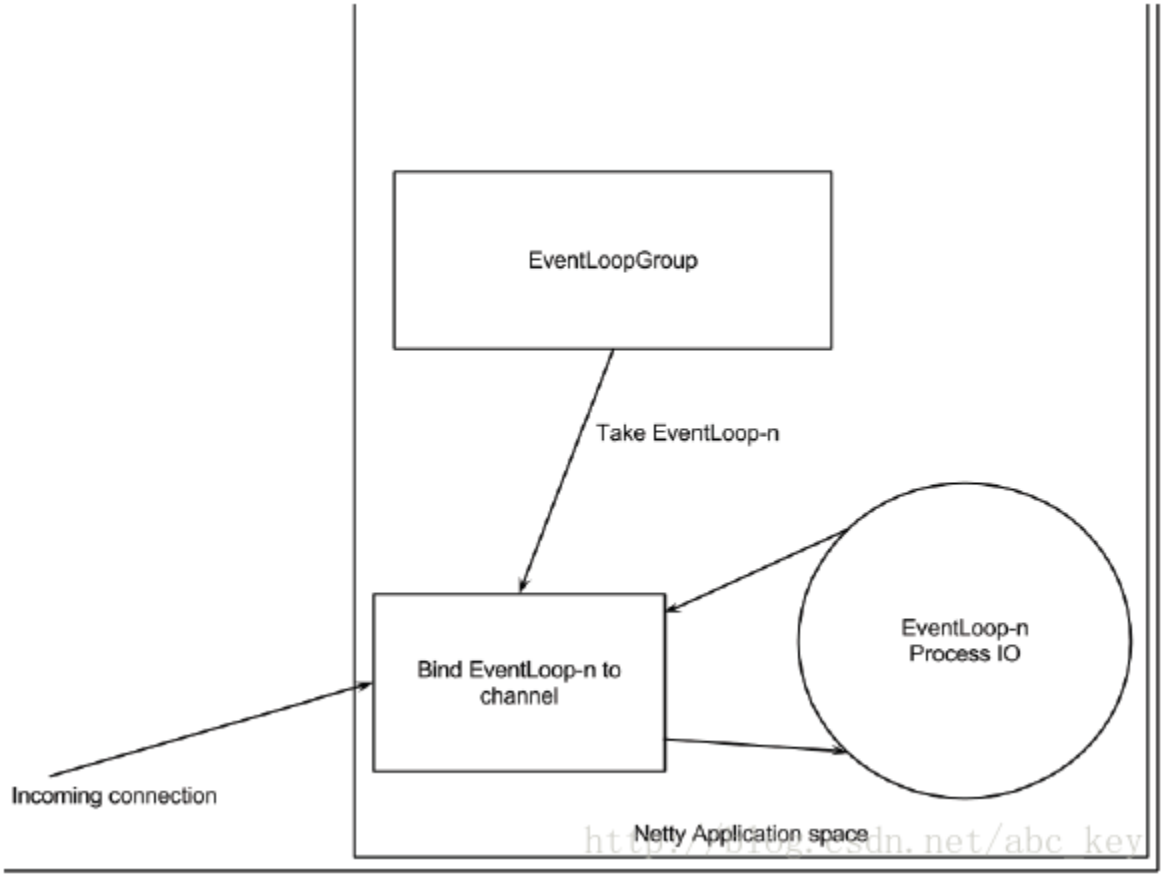
所有的 Netty 程序都是基于 ChannelPipeline。ChannelPipeline 和 EventLoop 和 EventLoopGroup 密切相关，因为它们三个都和事件处理相关，所以这就是为什么它们处理 IO 的工作由 EventLoop 管理的原因。

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)

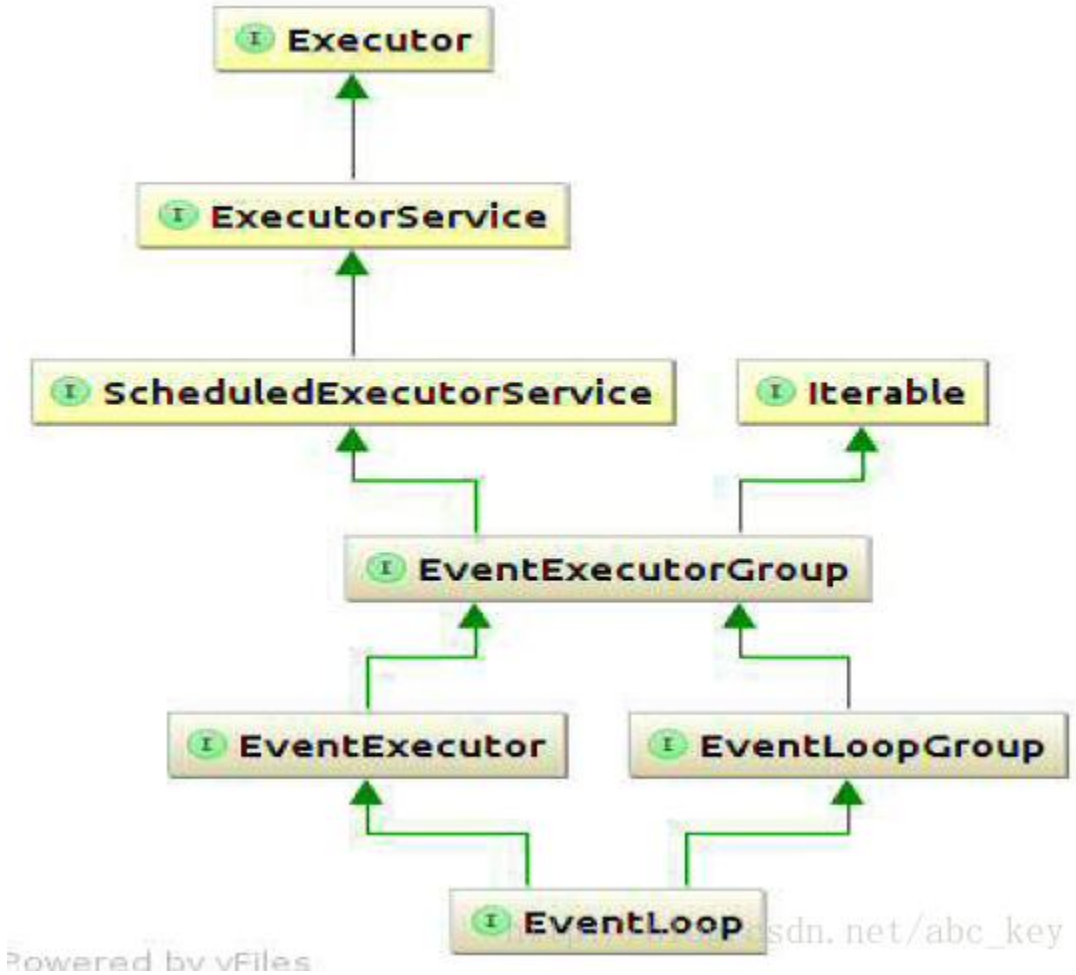
Netty 是一个非阻塞、事件驱动的网络框架。Netty 实际上是使用多线程处理 IO 事件，对于熟悉多线程编程的读者可能会需要同步代码。这样的方式不好，因为同步会影响程序的性能，Netty 的设计保证程序处理事件不会有同步。

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”。

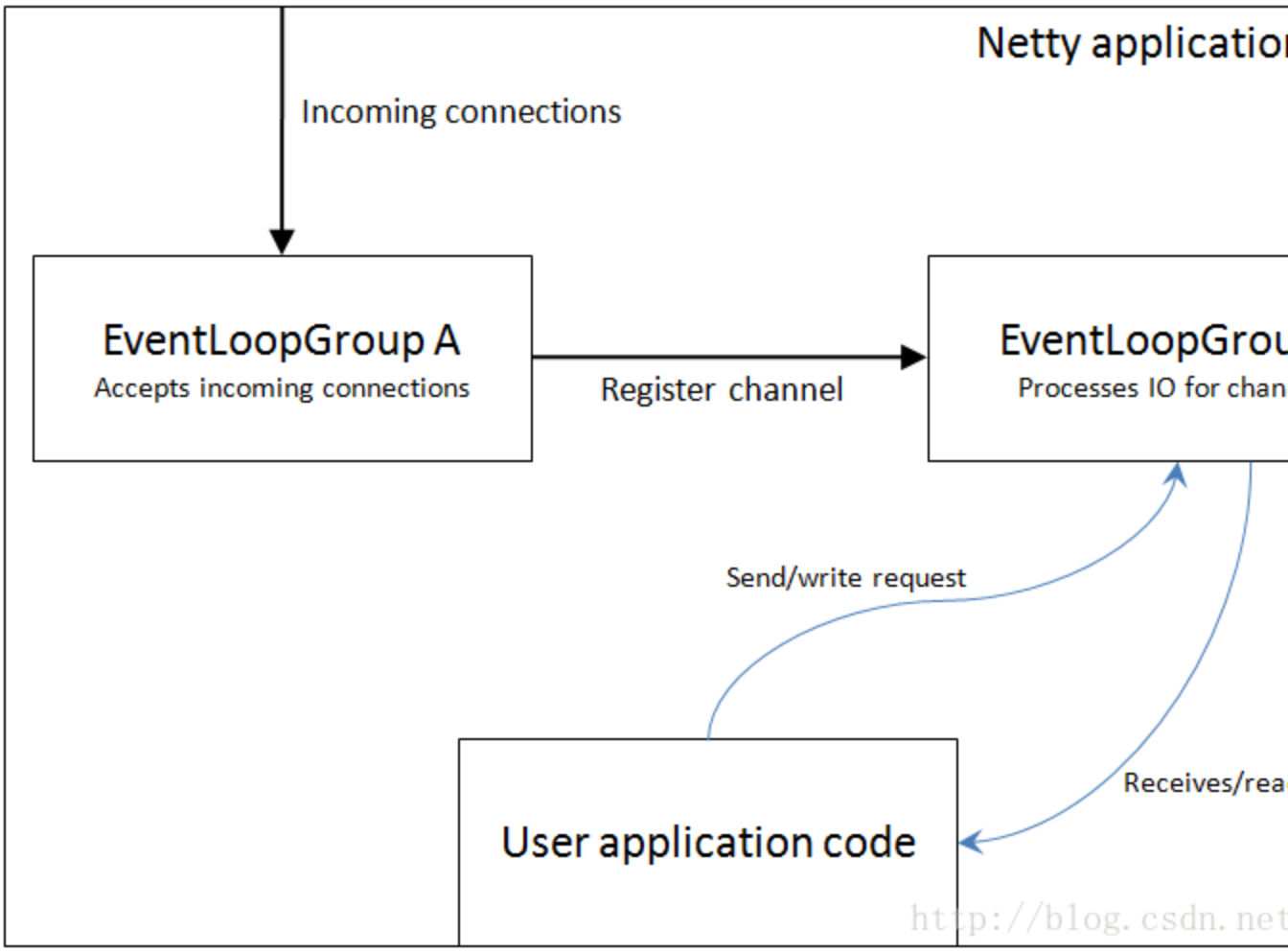
两种 bootsstraps 之间有一些相似之处，其实他们有很多相似之处，也有一些不同。Bootstrap 和 ServerBootstrap 之间的差异：

- 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：

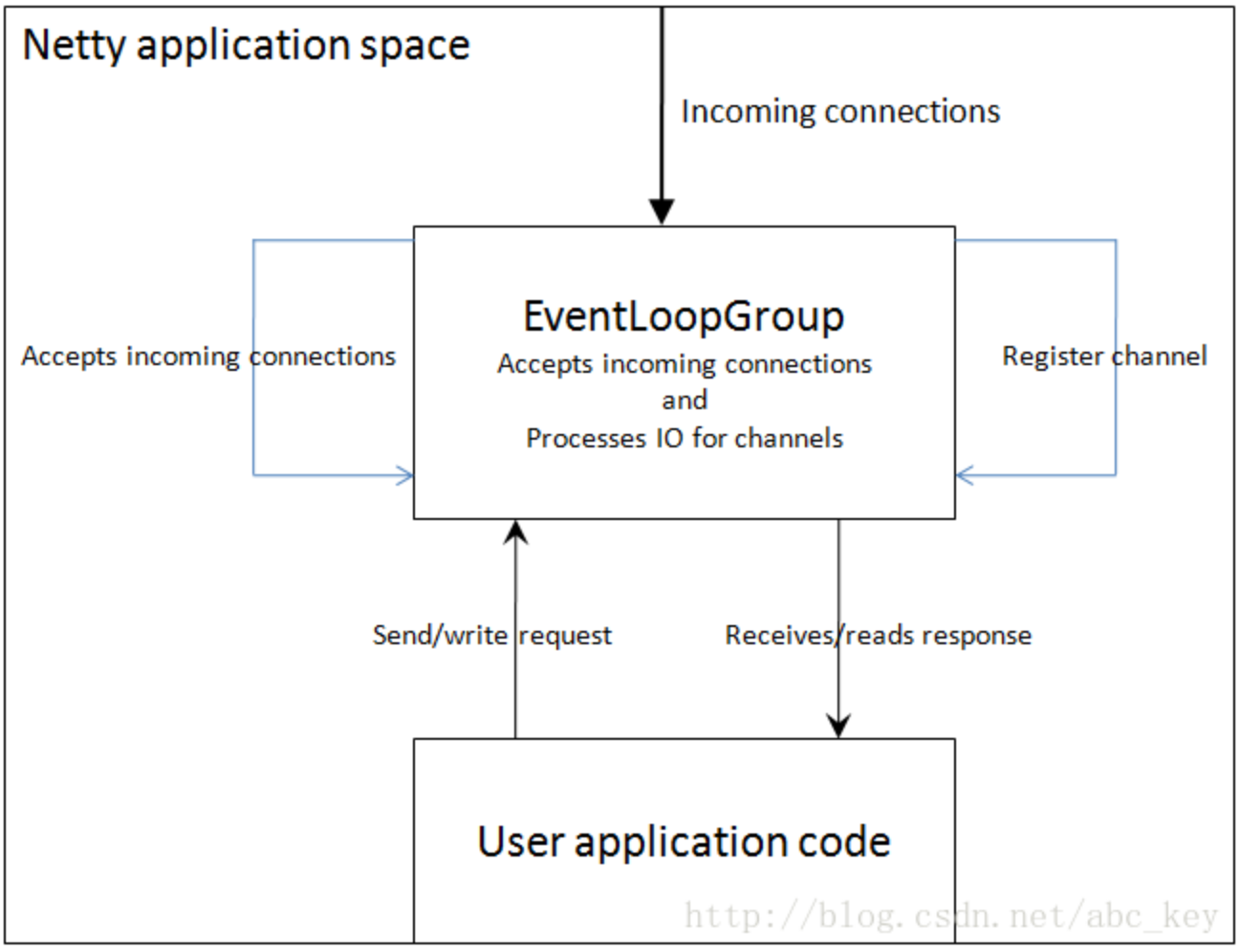


上图中，EventLoopGroup A 唯一的目的就是接受连接然后交给 EventLoopGroup B。Netty 可以使用两个不同的 Group，因为服务器程序需要接受很多客户端连接的情况下，一个 EventLoopGroup 将是程序性能的瓶颈，因为事件循环忙于处理连接请求，没有多余的资源和空闲来处理业务逻辑，最后的结果会是很多连接请求超时。若有两 EventLoops， 即使在

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。

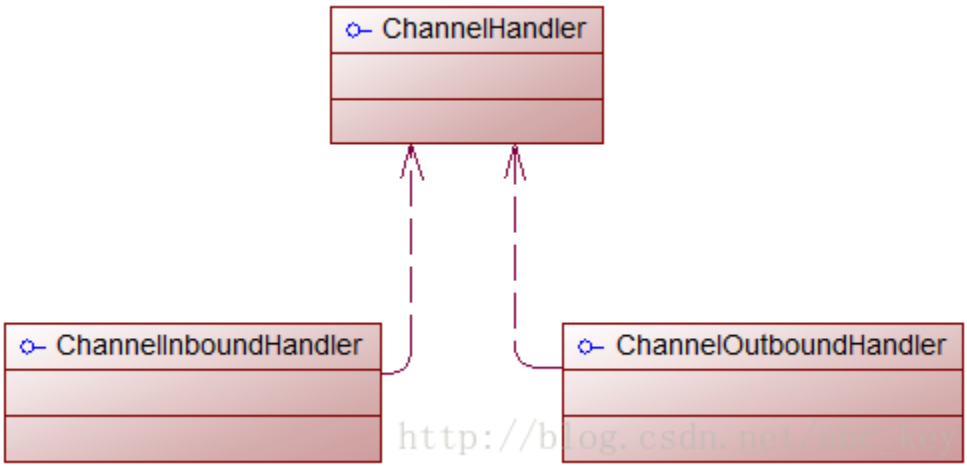
下一节我们将介绍 Netty 是如何执行 IO 操作以及在什么时候执行。

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 wirte 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。即使你没有意思到这一点，若果你使用 Netty 应用将至少有一个 ChannelHandler 参与，换句话说，ChannelHandler 对很多事情是关键的。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：

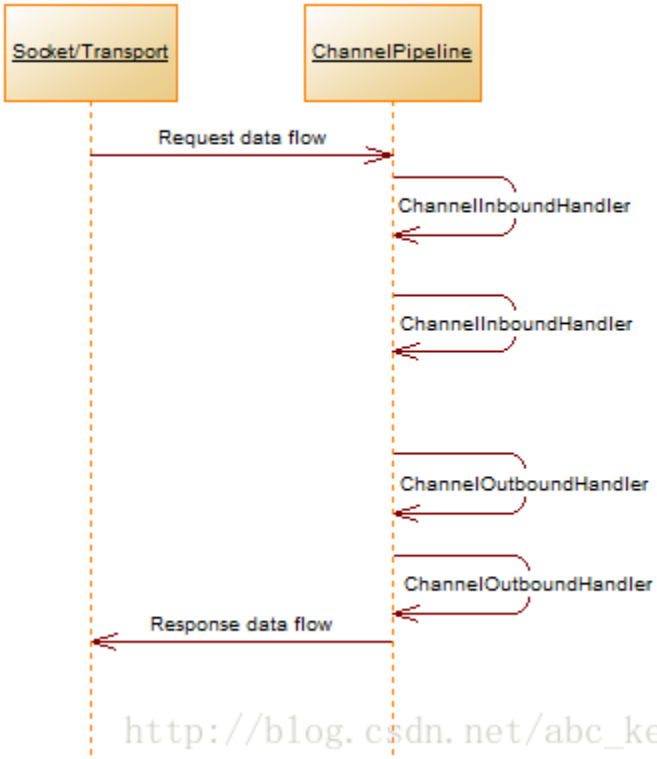


上图显示的比较容易，更重要的是 ChannelHandler 在数据流方面的应用，在这里讨论的例子只是一个简单的例子。ChannelHandler 被应用在许多方面，在本书中会慢慢学习。

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做的处理。

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*Adapter 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 涨，我们重点看看 ecoders,decoders 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。

第四章：Transports(传输)

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] view plaincopy

```
1. package netty.in.action;
2.
3. import java.io.IOException;
4. import java.io.OutputStream;
5. import java.net.ServerSocket;
6. import java.net.Socket;
7. import java.nio.charset.Charset;
8.
9. /**
10. * Blocking networking without Netty
11. * @author c.k
12. *
13. */
14. public class PlainOioServer {
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 " + clientSocke
    t);
24.                 //create new thread to handle connection
25.                 new Thread(new Runnable() {
26.                     @Override
27.                     public void run() {
28.                         OutputStream out;
29.                         try{
30.                             out = clientSocket.getOutputStream();
31.                             //write message to connected client
32.                             out.write("Hi!\r\n".getBytes(Charset.forName("UT
    F-8")));
33.                             out.flush();
34.                             //close connection once message written and flus
    hed
35.                             clientSocket.close();
36.                         }catch(IOException e){
37.                             try {
38.                                 clientSocket.close();
39.                             } catch (IOException e1) {
40.                                 e1.printStackTrace();
41.                             }
42.                         }
43.                     }
44.                 }).start();//start thread to begin handling
45.             }
46.         }catch(Exception e){
47.             e.printStackTrace();
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很严重的问题，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] view plaincopy

```java
package netty.in.action;

import java.net.InetSocketAddress;
import java.net.ServerSocket;
import java.nio.ByteBuffer;
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;
import java.util.Iterator;
/**
 * Asynchronous networking without Netty
 * @author c.k
 *
 */
public class PlainNioServer {

    public void server(int port) throws Exception {
        System.out.println("Listening for connections on port " + port);
        //open Selector that handles channels
        Selector selector = Selector.open();
        //open ServerSocketChannel
        ServerSocketChannel serverChannel = ServerSocketChannel.open();
        //get ServerSocket
        ServerSocket serverSocket = serverChannel.socket();
        //bind server to port
        serverSocket.bind(new InetSocketAddress(port));
        //set to non-blocking
        serverChannel.configureBlocking(false);
        //register ServerSocket to selector and specify that it is intereste
    d in new accepted clients
        serverChannel.register(selector, SelectionKey.OP_ACCEPT);
        final ByteBuffer msg = ByteBuffer.wrap("Hi!\r\n".getBytes());
        while (true) {
            //Wait for new events that are ready for process. This will bloc
    k until something happens
            int n = selector.select();
            if (n > 0) {
                //Obtain all SelectionKey instances that received events
```

```
38.                 Iterator<SelectionKey> iter = selector.selectedKeys().iterat
    or();
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 = (ServerSocketChanne
    l) 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 writ
    e data
53.                         if (key.isWritable()) {
54.                             SocketChannel client = (SocketChannel) key.chann
    el();
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] view plaincopy

```
1. package netty.in.action;
2.
3. import java.net.InetSocketAddress;
4.
5. import io.netty.bootstrap.ServerBootstrap;
6. import io.netty.buffer.ByteBuf;
7. import io.netty.buffer.Unpooled;
8. import io.netty.channel.Channel;
9. 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.oio.OioServerSocketChannel;
17. import io.netty.util.CharsetUtil;
18.
19. public class NettyOioServer {
20.
21.     public void server(int port) throws Exception {
22.         final ByteBuf buf = Unpooled.unreleasableBuffer(Unpooled.copiedBuffe
    r("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 OIO 阻塞模式
29.             b.group(group).channel(OioServerSocketChannel.class).localAddres
    s(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 Except
    ion {
34.                             //添加一个“入站”handler 到 ChannelPipeline
```

```
                            ch.pipeline().addLast(new ChannelInboundHandlerA
dapter() {
                                @Override
                                public void channelActive(ChannelHandlerCont
ext ctx) throws Exception {
                                    //连接后，写消息到客户端，写完后便关闭连接
                                    ctx.writeAndFlush(buf.duplicate()).addLi
stener(ChannelFutureListener.CLOSE);
                                }
                            });
                        }
                    });
            //绑定服务器接受连接
            ChannelFuture f = b.bind().sync();
            f.channel().closeFuture().sync();
        } catch (Exception e) {
            //释放所有资源
            group.shutdownGracefully();
        }
    }

}
```

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] view plaincopy

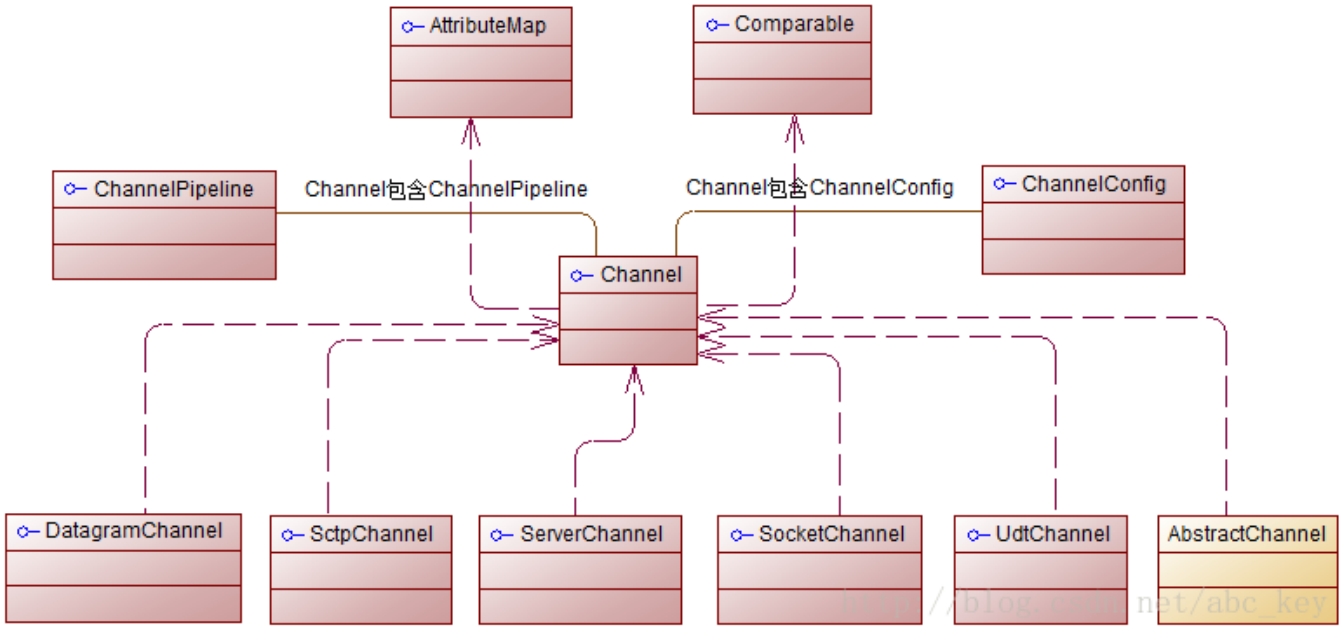
```
package netty.in.action;

import io.netty.bootstrap.ServerBootstrap;
import io.netty.buffer.ByteBuf;
import io.netty.buffer.Unpooled;
import io.netty.channel.ChannelFuture;
import io.netty.channel.ChannelFutureListener;
import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.ChannelInboundHandlerAdapter;
import io.netty.channel.ChannelInitializer;
import io.netty.channel.EventLoopGroup;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.SocketChannel;
import io.netty.channel.socket.nio.NioServerSocketChannel;
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.copiedBuffe
    r("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 异步模式
29.             b.group(group).channel(NioServerSocketChannel.class).localAddres
    s(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 ChannelInboundHandlerA
    dapter() {
36.                                 @Override
37.                                 public void channelActive(ChannelHandlerCont
    ext 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()).addLi
    stener(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] view plaincopy

```java
Channel channel = ...
//Create ByteBuf that holds data to write
ByteBuf buf = Unpooled.copiedBuffer("your data", CharsetUtil.UTF_8);
//Write data
ChannelFuture cf = channel.write(buf);
//Add ChannelFutureListener to get notified after write completes
cf.addListener(new ChannelFutureListener() {
    @Override
    public void operationComplete(ChannelFuture future) {
        //Write operation completes without error
        if (future.isSuccess()) {
            System.out.println(.Write successful.);
        } else {
            //Write operation completed but because of error
            System.err.println(.Write error.);
            future.cause().printStacktrace();
        }
    }
});
```

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] view plaincopy

```java
final Channel channel = ...
//Create ByteBuf that holds data to write
final ByteBuf buf = Unpooled.copiedBuffer("your data",CharsetUtil.UTF_8);
//Create Runnable which writes data to channel
Runnable writer = new Runnable() {
    @Override
    public void run() {
        channel.write(buf.duplicate());
    }
};
//Obtain reference to the Executor which uses threads to execute tasks
Executor executor = Executors.newChachedThreadPool();
// write in one thread
//Hand over write task to executor for execution in thread
executor.execute(writer);
// write in another thread
//Hand over another write task to executor for execution in thread
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

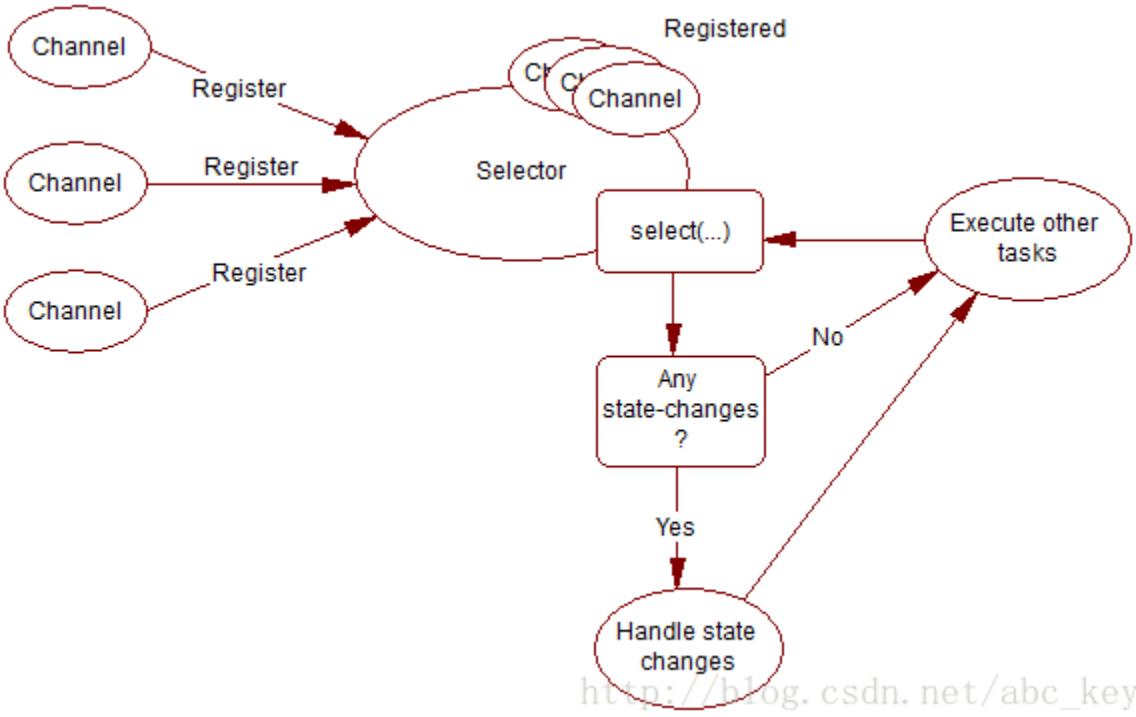
NIO 传输是目前最常用的方式，它通过使用选择器提供了完全异步的方式操作所有的 I/O，NIO 从 Java 1.4 才被提供。NIO 中，我们可以注册一个通道或获得某个通道的改变的状态，通道状态有下面几种改变：

- 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

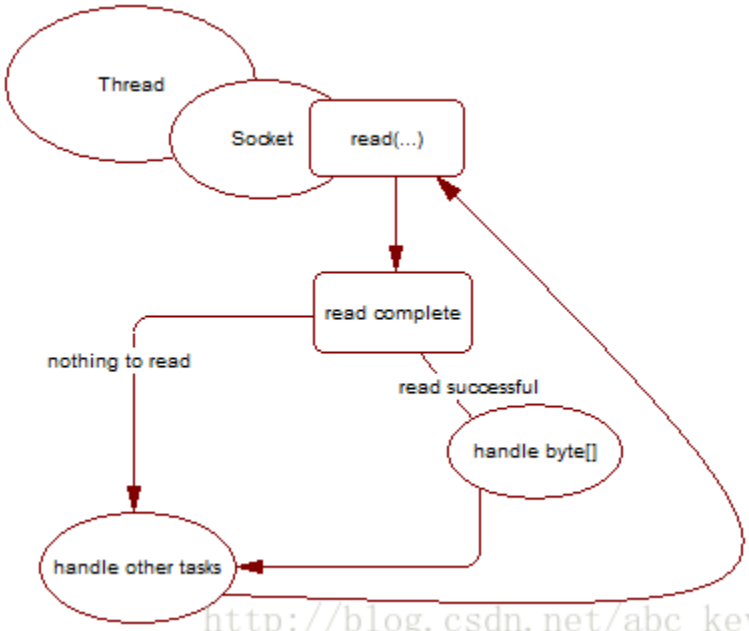
Netty 中的 NIO 传输就是基于这样的模型来接收和发送数据，通过封装将自己的接口提供给用户使用，这完全隐藏了内部实现。如前面所说，Netty 隐藏内部的实现细节，将抽象出来的 API 暴露出来供使用，下面是处理流程图：



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

第五章：Buffers(缓冲)

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] view plaincopy

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

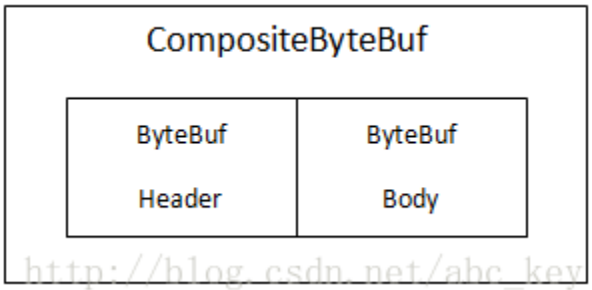
访问直接缓冲区的数据数组需要更多的编码和更复杂的操作，建议若需要在数组访问数据使用堆缓冲区会更好。

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] view plaincopy

```java
CompositeByteBuf compBuf = Unpooled.compositeBuffer();
ByteBuf heapBuf = Unpooled.buffer(8);
ByteBuf directBuf = Unpooled.directBuffer(16);
//添加 ByteBuf 到 CompositeByteBuf
compBuf.addComponents(heapBuf,directBuf);
//删除第一个 ByteBuf
compBuf.removeComponent(0);
Iterator<ByteBuf> iter = compBuf.iterator();
while(iter.hasNext()){
    System.out.println(iter.next().toString());
}
//使用数组访问数据
if(!compBuf.hasArray()){
    int len = compBuf.readableBytes();
    byte[] arr = new byte[len];
    compBuf.getBytes(0, arr);
}
```

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] view plaincopy

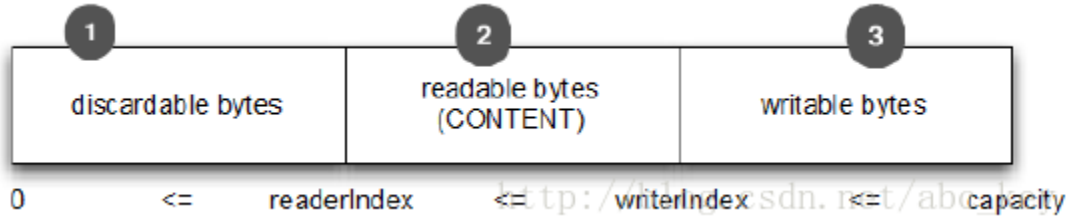
```java
//create a ByteBuf of capacity is 16
ByteBuf buf = Unpooled.buffer(16);
//write data to buf
for(int i=0;i<16;i++){
    buf.writeByte(i+1);
}
//read data from buf
for(int i=0;i<buf.capacity();i++){
    System.out.println(buf.getByte(i));
}
```

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

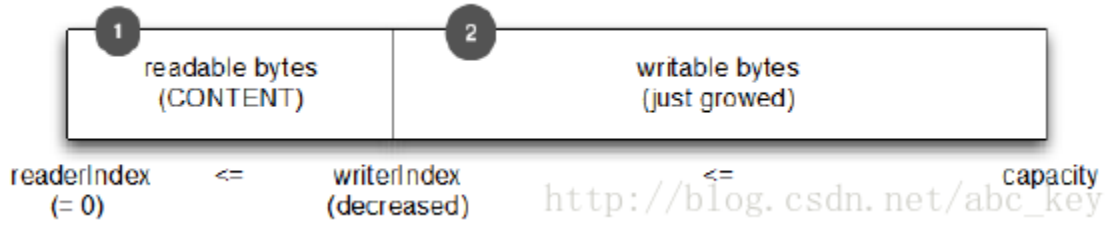
ByteBuf 提供两个指针变量支付读和写操作，读操作是使用 readerIndex()，写操作时使用 writerIndex()。这和 JDK 的 ByteBuffer 不同，ByteBuffer 只有一个方法来设置索引，所以需要使用 flip()方法来切换读和写模式。

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 IndexOutOfBoundException。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] view plaincopy

```java
ByteBuf buf = Unpooled.buffer(16);
while(buf.isReadable()){
    System.out.println(buf.readByte());
}
```

(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 IndexOutOfBoundException。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：

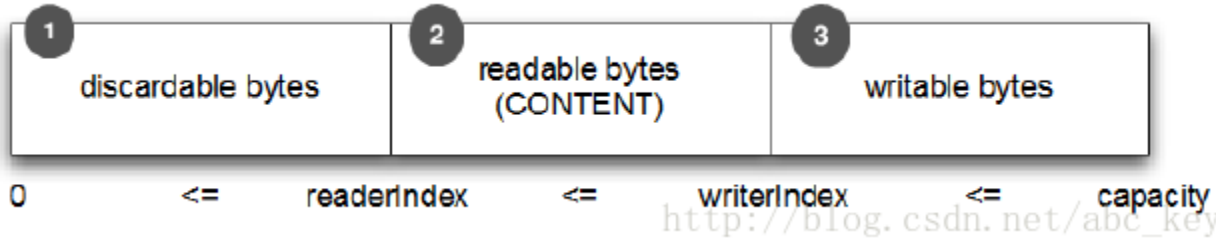
[java] view plaincopy

```java
Random random = new Random();
ByteBuf buf = Unpooled.buffer(16);
while(buf.writableBytes() >= 4){
    buf.writeInt(random.nextInt());
}
```

5.3.6 清除缓冲区索引 Clearing the buffer indexs

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 IndexOutOfBoundException。

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] view plaincopy

```
1.  // get a Charset of UTF-8
2.  Charset utf8 = Charset.forName("UTF-8");
```

```
3.  // get a ByteBuf
4.  ByteBuf buf = Unpooled.copiedBuffer("“Netty in Action rocks!“", utf8);
5.  // slice
6.  ByteBuf sliced = buf.slice(0, 14);
7.  // copy
8.  ByteBuf copy = buf.copy(0, 14);
9.  // 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] view plaincopy

```
1.  ServerBootstrap b = new ServerBootstrap();
```

```
2.  b.group(group).channel(NioServerSocketChannel.class).localAddress(new InetSo
    cketAddress(port))
3.          .childHandler(new ChannelInitializer<SocketChannel>() {
4.              @Override
5.              protected void initChannel(SocketChannel ch) throws Exception {

6.                  // get ByteBufAllocator instance by Channel.alloc()
7.                  ByteBufAllocator alloc0 = ch.alloc();
8.                  ch.pipeline().addLast(new ChannelInboundHandlerAdapter() {
9.                      @Override
10.                     public void channelActive(ChannelHandlerContext ctx) thr
    ows Exception {
11.                         //get ByteBufAllocator instance by ChannelHandlerCon
    text.alloc()
12.                         ByteBufAllocator alloc1 = ctx.alloc();
13.                         ctx.writeAndFlush(buf.duplicate()).addListener(Chann
    elFutureListener.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] view plaincopy

```
1.  //创建复合缓冲区
2.  CompositeByteBuf compBuf = Unpooled.compositeBuffer();
3.  //创建堆缓冲区
4.  ByteBuf heapBuf = Unpooled.buffer(8);
5.  //创建直接缓冲区
6.  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。

第六章：ChannelHandler

本章介绍

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

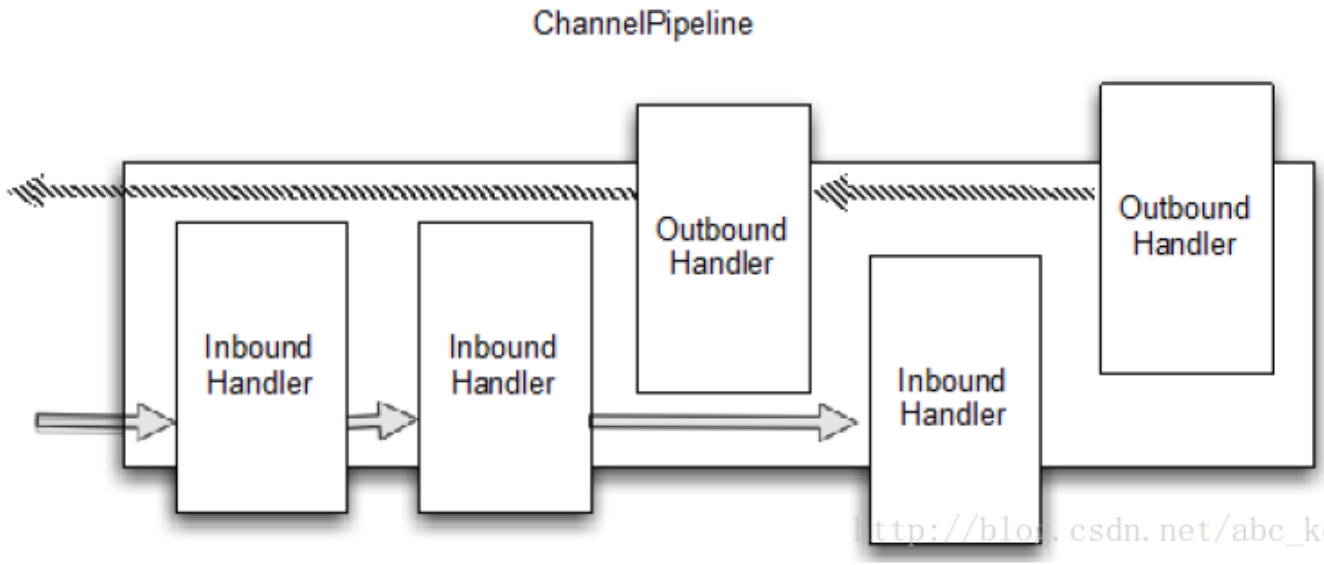
接受连接或创建他们只是你的应用程序的一部分，虽然这些任何很重要，但是一个网络应用程序旺旺是更复杂的，需要更多的代码编写，如处理传入和传出的数据。Netty 提供了一个强大的处理这些事情的功能，允许用户自定义 ChannelHandler 的实现来处理数据。使得 ChannelHandler 更强大的是可以连接每个 ChannelHandler 来实现任务，这有助于代码的整洁和重用。但是处理数据只是 ChannelHandler 所做的事情之一，也可以压制 I/O 操作，例如写请求。所有这些都可以动态实现。

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] view plaincopy

```
1. ChannelPipeline pipeline = ch.pipeline();
2. FirstHandler firstHandler = new FirstHandler();
3. pipeline.addLast("handler1", firstHandler);
4. pipeline.addFirst("handler2", new SecondHandler());
5. pipeline.addLast("handler3", new ThirdHandler());
6. pipeline.remove("“handler3“");
7. pipeline.remove(firstHandler);
8. 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。

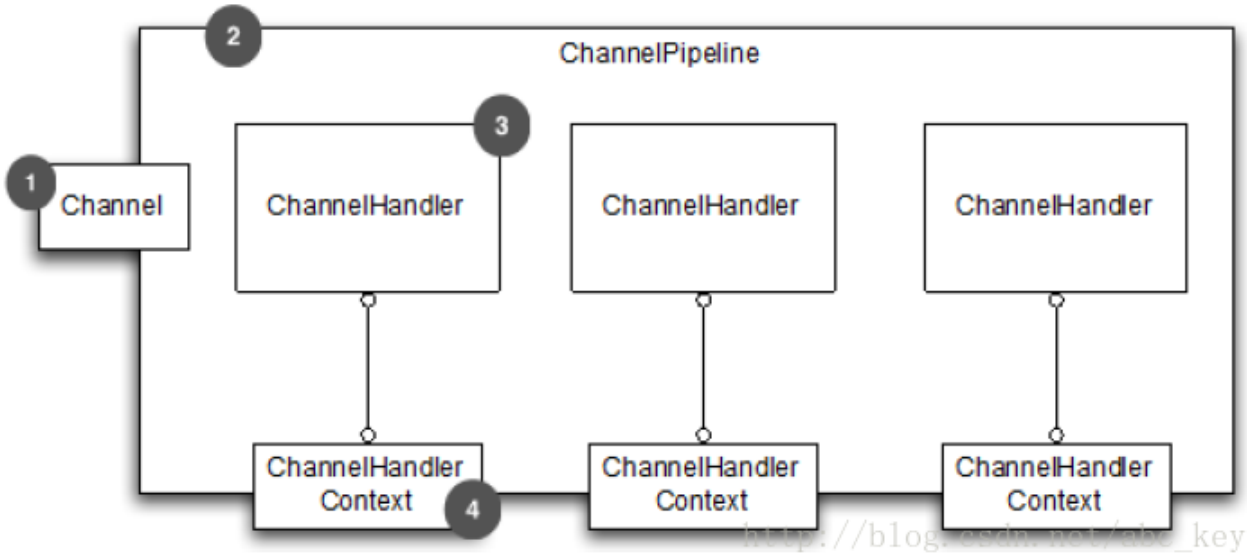
ChannelPipeline 除了一些修改的方法，还有很多其他的方法，具体是方法及使用可以看 API 文档或源码。

6.2 ChannelHandlerContext

每个 ChannelHandler 被添加到 ChannelPipeline 后，都会创建一个 ChannelHandlerContext 并与之创建的 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

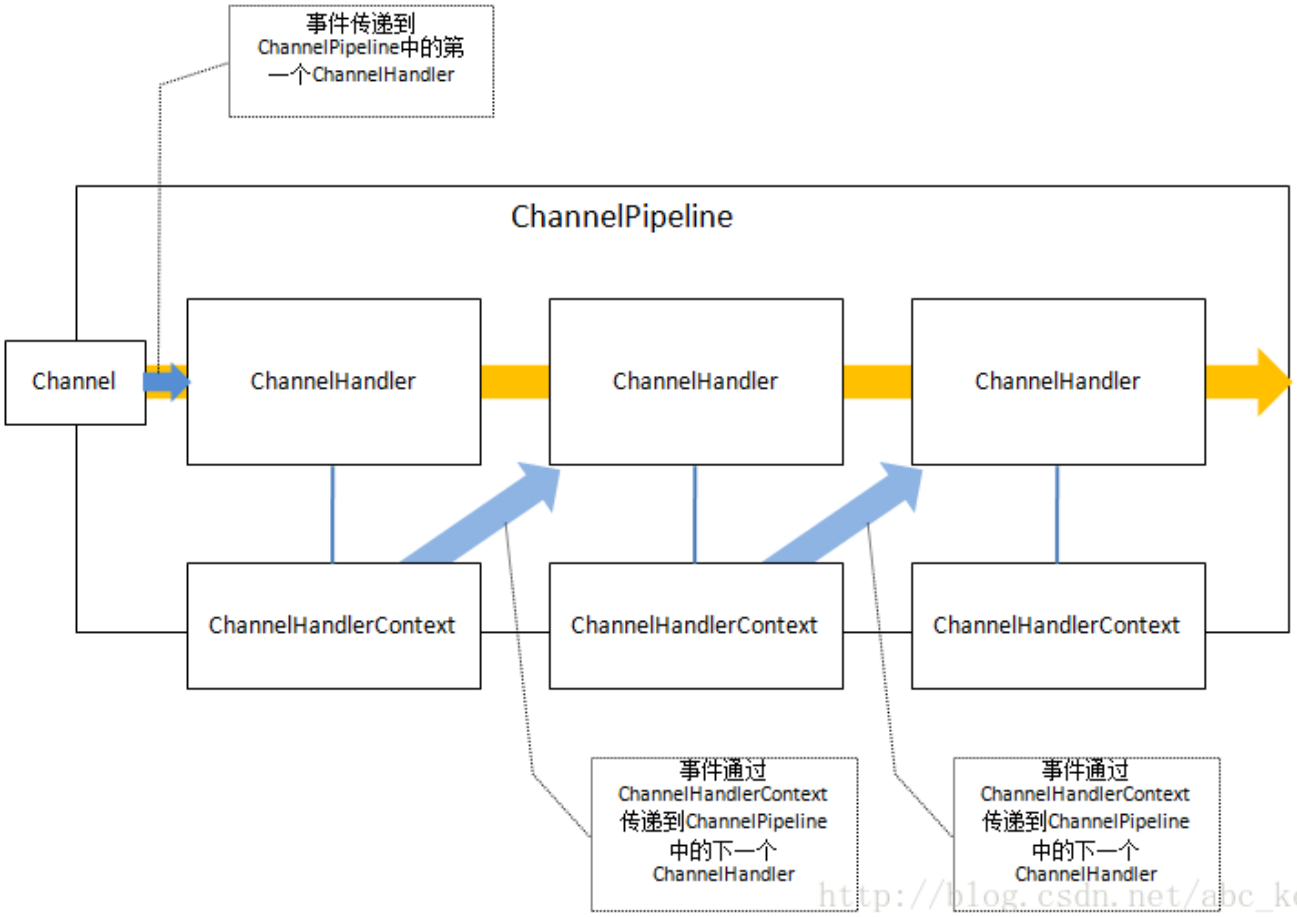
这两个方法都可以让事件流全部通过 ChannelPipeline。无论从头部还是尾部开始，因为它主要依赖于事件的性质。如果是一个“入站”事件，它开始于头部；若是一个“出站”事件，则开始于尾部。

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

[java] view plaincopy

```java
@Override
protected void initChannel(SocketChannel ch) throws Exception {
    ch.pipeline().addLast(new ChannelInboundHandlerAdapter() {
        @Override
        public void channelActive(ChannelHandlerContext ctx) throws Exception {
            //Event via Channel
            Channel channel = ctx.channel();
            channel.write(Unpooled.copiedBuffer("netty in action", CharsetUtil.UTF_8));
            //Event via ChannelPipeline
            ChannelPipeline pipeline = ctx.pipeline();
            pipeline.write(Unpooled.copiedBuffer("netty in action", CharsetUtil.UTF_8));
        }
    });
}
```

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：

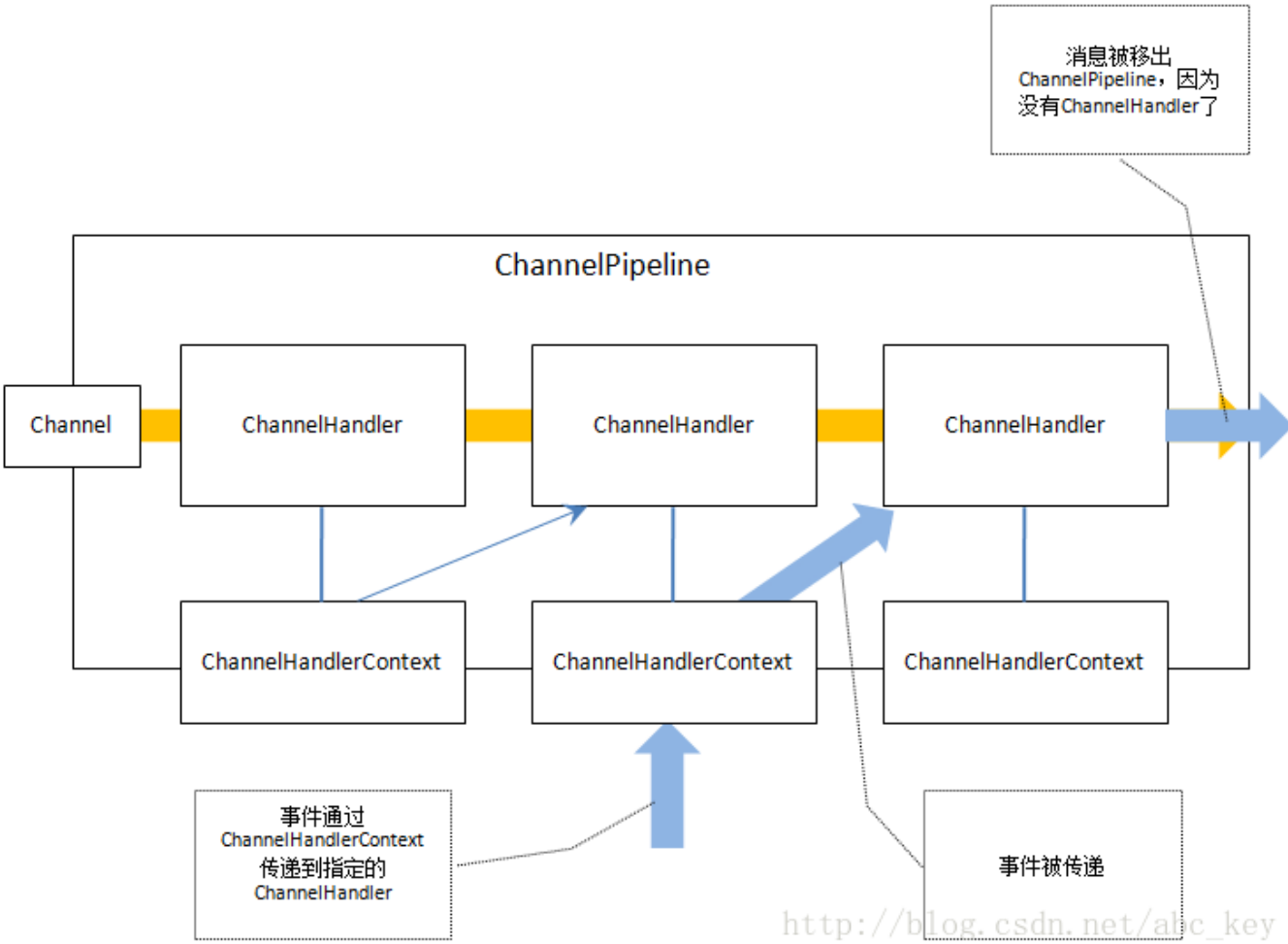
- 为了节省开销，不感兴趣的 ChannelHandler 不让通过
- 排除一些 ChannelHandler

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] view plaincopy

```
1. // Get reference of ChannelHandlerContext
2. ChannelHandlerContext ctx = ..;
3. // Write buffer via ChannelHandlerContext
4. 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] view plaincopy

```
1.  public class WriteHandler extends ChannelHandlerAdapter {
2.      private ChannelHandlerContext ctx;
3.
4.      @Override
5.      public void handlerAdded(ChannelHandlerContext ctx) throws Exception {
```

```
6.          this.ctx = ctx;
7.      }
8.
9.      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] view plaincopy

```
1.  @Sharable
2.  public class NotSharableHandler extends ChannelInboundHandlerAdapter {
3.
4.      private int count;
5.
6.      @Override
7.      public void channelRead(ChannelHandlerContext ctx, Object msg) throws Ex
    ception {
8.          count++;
9.          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。

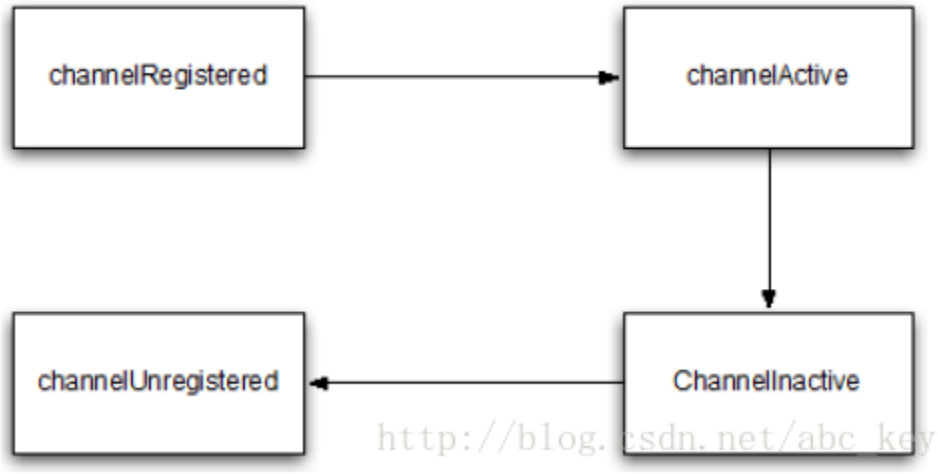
为什么要共享 ChannelHandler？使用@Sharable 注解共享一个 ChannelHandler 在一些需求中还是有很好的作用的，如使用一个 ChannelHandler 来统计连接数或来处理一些全局数据等等。

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：

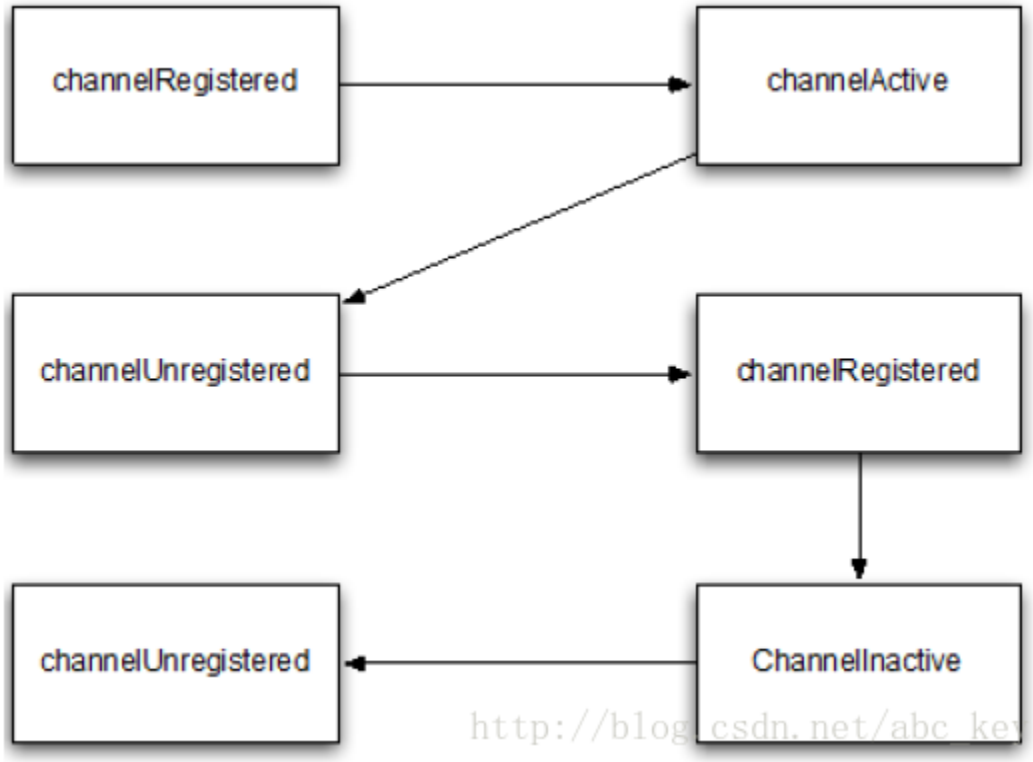
- channelUnregistered
- channelRegistered
- channelActive
- channelInactive

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：



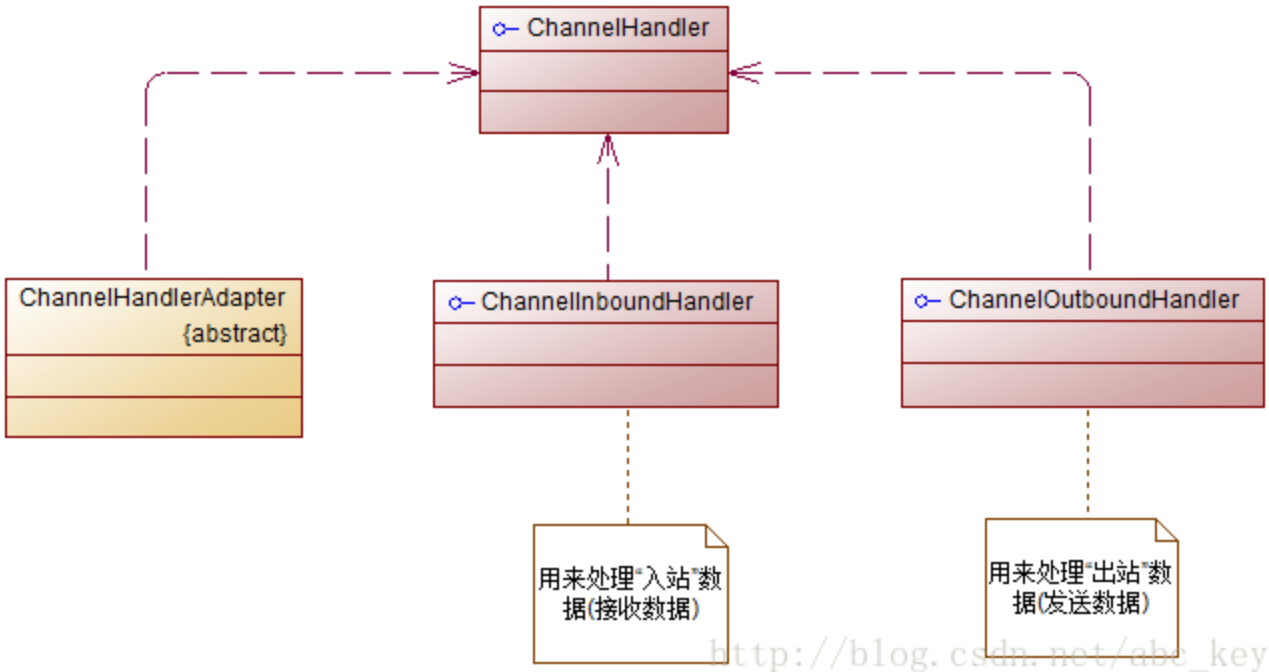
还可以看到额外的状态变化，因为用户允许从 EventLoop 中注销 Channel 暂停事件执行，然后再重新注册。在这种情况下，你会看到多个 channelRegistered 和 channelUnregistered 状态的变化，而永远只有一个 channelActive 和 channelInactive 的状态，因为一个通道在其生命周期内只能连接一次，之后就会被回收；重新连接，则是创建一个新的通道。

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
- channelInactive，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
/**
 * 实现ChannelInboundHandlerAdapter 的Handler，不会自动释放接收的消息对象
 * @author c.k
 *
 */
public class DiscardHandler extends ChannelInboundHandlerAdapter {
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {
        //手动释放消息
        ReferenceCountUtil.release(msg);
    }
}
```

[java] view plaincopy

```java
/**
 * 继承SimpleChannelInboundHandler，会自动释放消息对象
 * @author c.k
 *
 */
public class SimpleDiscardHandler extends SimpleChannelInboundHandler<Object> {
    @Override
    protected void channelRead0(ChannelHandlerContext ctx, Object msg) throws Exception {
        //不需要手动释放
    }
}
```

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] view plaincopy

```java
public class DiscardOutboundHandler extends ChannelOutboundHandlerAdapter {

    @Override
    public void write(ChannelHandlerContext ctx, Object msg, ChannelPromise promise) throws Exception {
        ReferenceCountUtil.release(msg);
        promise.setSuccess();
    }
}
```

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。

第七章：编解码器 Codec

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“入站”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，

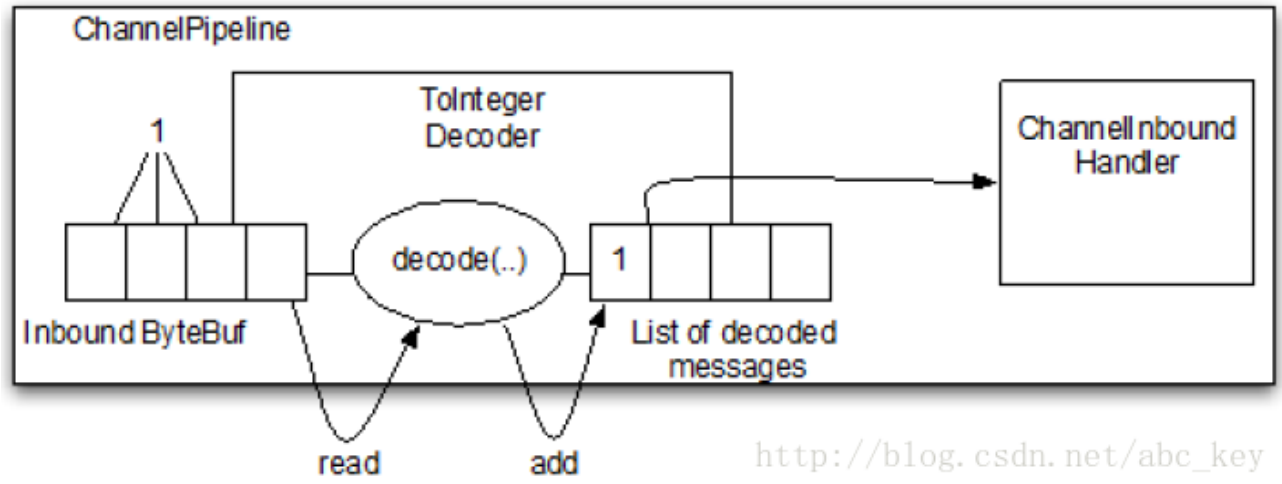
就是将入站数据转换格式后传递到 ChannelPipeline 中的下一个 ChannelInboundHandler 进行处理；这样的处理时很灵活的，我们可以将解码器放在 ChannelPipeline 中，重用逻辑。

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“入站”ByteBuf 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] view plaincopy

```java
/**
 * Integer 解码器,ByteToMessageDecoder 实现
 * @author c.k
 *
 */
public class ToIntegerDecoder extends ByteToMessageDecoder {

    @Override
```

```
9.      protected void decode(ChannelHandlerContext ctx, ByteBuf 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 ByteBuf 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 ByteBuf 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 ByteBuf 支持，如果调用一个不支持的操作会抛出 DecoderException。
- ByteBuf.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] view plaincopy

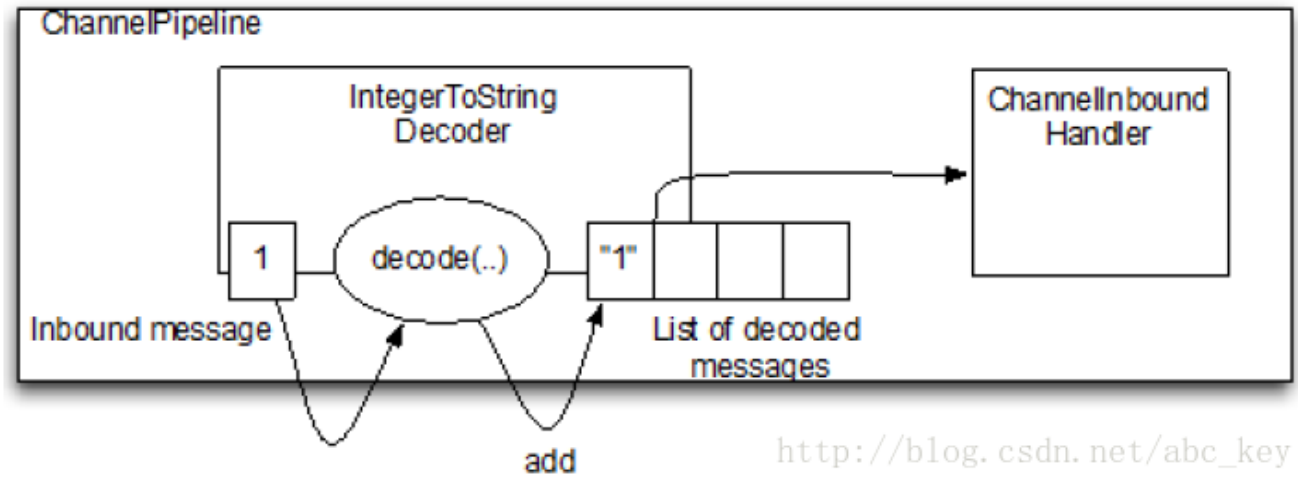
```
1.  /**
2.   * Integer 解码器,ReplayingDecoder 实现
3.   * @author c.k
4.   *
5.   */
6.  public class ToIntegerReplayingDecoder extends ReplayingDecoder<Void> {
7.
8.      @Override
9.      protected void decode(ChannelHandlerContext ctx, ByteBuf in, List<Object
    > out) throws Exception {
10.         out.add(in.readInt());
11.     }
12. }
```

当从接收的数据 ByteBuf 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] view plaincopy

```java
/**
 * 将接收的Integer 消息转成 String 类型，MessageToMessageDecoder 实现
 * @author c.k
 *
 */
public class IntegerToStringDecoder extends MessageToMessageDecoder<Integer>
 {

    @Override
    protected void decode(ChannelHandlerContext ctx, Integer msg, List<Objec
t> out) throws Exception {
        out.add(String.valueOf(msg));
    }
}
```

7.2.4 解码器总结

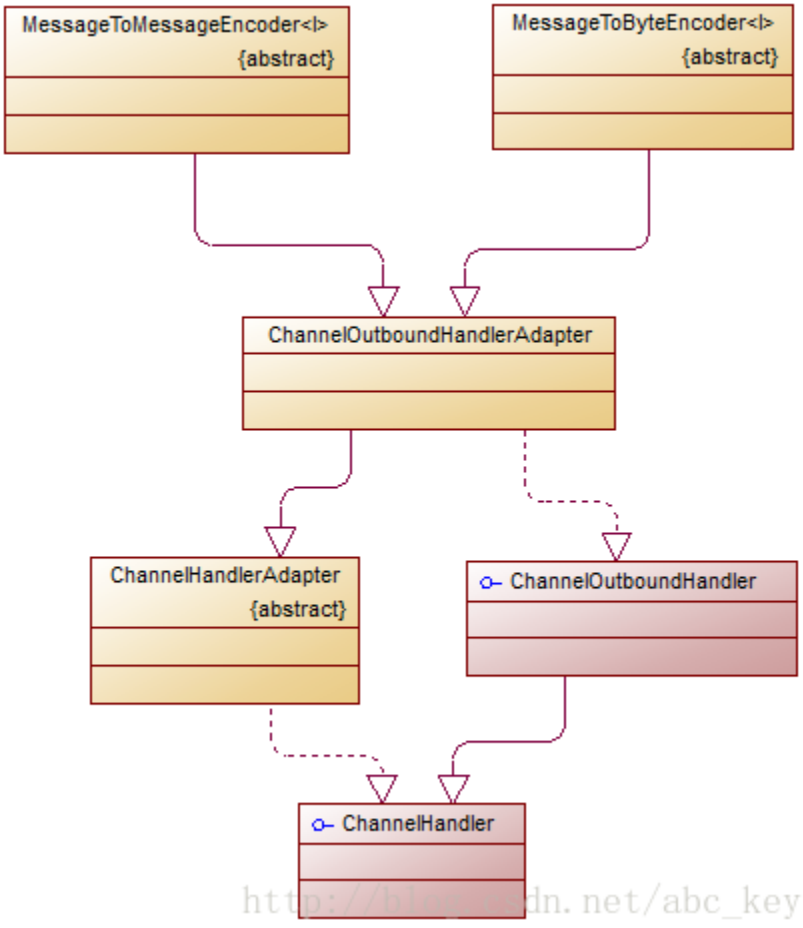
解码器是用来处理入站数据，Netty 提供了很多解码器的实现，可以根据需求详细了解。那我们发送数据需要将数据编码，Netty 中也提供了编码器的支持。下一节将讲解如何实现编码器。

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：

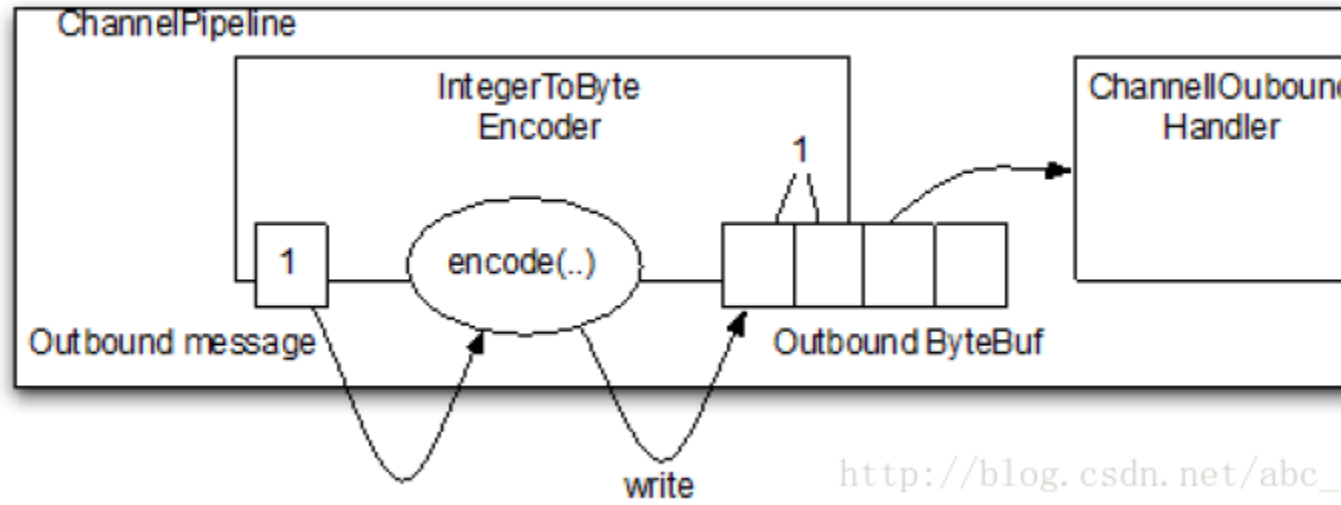
- 消息对象编码成消息对象
- 消息对象编码成字节码

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：



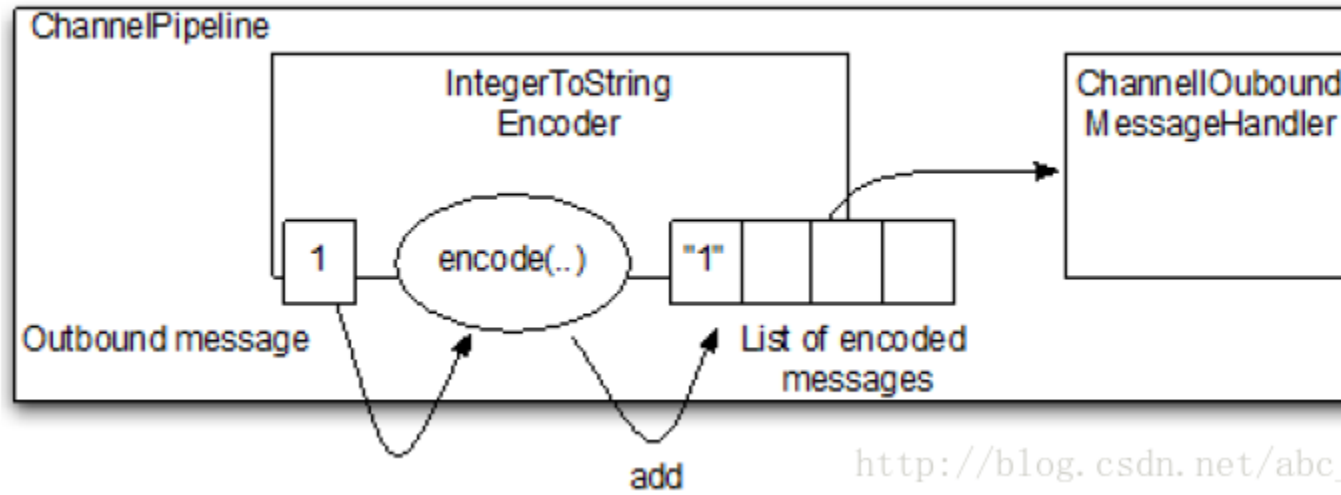
实现代码如下：

[java] view plaincopy

```java
/**
 * 编码器，将 Integer 值编码成 byte[]，MessageToByteEncoder 实现
 * @author c.k
 *
 */
public class IntegerToByteEncoder extends MessageToByteEncoder<Integer> {
    @Override
    protected void encode(ChannelHandlerContext ctx, Integer msg, ByteBuf ou
t) throws Exception {
        out.writeInt(msg);
    }
}
```

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] view plaincopy

```java
/**
 * 编码器，将 Integer 编码成 String，MessageToMessageEncoder 实现
 * @author c.k
 *
 */
public class IntegerToStringEncoder extends MessageToMessageEncoder<Integer>
 {

    @Override
    protected void encode(ChannelHandlerContext ctx, Integer msg, List<Objec
t> out) throws Exception {
        out.add(String.valueOf(msg));
    }
}
```

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] view plaincopy

```java
public class ByteArrayDecoder extends MessageToMessageDecoder<ByteBuf> {
    @Override
    protected void decode(ChannelHandlerContext ctx, ByteBuf msg, List<Objec
t> out) throws Exception {
         // copy the ByteBuf content to a byte array
        byte[] array = new byte[msg.readableBytes()];
        msg.getBytes(0, array);

        out.add(array);
    }
}
```

[java] view plaincopy

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

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] view plaincopy

```java
package netty.in.action;

import java.util.List;

import io.netty.buffer.ByteBuf;
import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.ChannelHandler.Sharable;
import io.netty.handler.codec.MessageToMessageCodec;
import io.netty.handler.codec.http.websocketx.BinaryWebSocketFrame;
import io.netty.handler.codec.http.websocketx.CloseWebSocketFrame;
import io.netty.handler.codec.http.websocketx.ContinuationWebSocketFrame;
import io.netty.handler.codec.http.websocketx.PingWebSocketFrame;
import io.netty.handler.codec.http.websocketx.PongWebSocketFrame;
import io.netty.handler.codec.http.websocketx.TextWebSocketFrame;
import io.netty.handler.codec.http.websocketx.WebSocketFrame;

@Sharable
public class WebSocketConvertHandler extends
        MessageToMessageCodec<WebSocketFrame, WebSocketConvertHandler.MyWebSocketFrame> {

    public static final WebSocketConvertHandler INSTANCE = new WebSocketConvertHandler();

    @Override
    protected void encode(ChannelHandlerContext ctx, MyWebSocketFrame msg, List<Object> out) throws Exception {
        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 " + m
    sg);
46.         }
47.     }
48.
49.     @Override
50.     protected void decode(ChannelHandlerContext ctx, WebSocketFrame msg, Lis
    t<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, m
    sg.content().copy()));
57.             return;
58.         }
59.         if (msg instanceof PingWebSocketFrame) {
60.             out.add(new MyWebSocketFrame(MyWebSocketFrame.FrameType.PING, ms
    g.content().copy()));
61.             return;
62.         }
63.         if (msg instanceof PongWebSocketFrame) {
```

```
64.             out.add(new MyWebSocketFrame(MyWebSocketFrame.FrameType.PONG, ms
    g.content().copy()));
65.             return;
66.         }
67.         if (msg instanceof TextWebSocketFrame) {
68.             out.add(new MyWebSocketFrame(MyWebSocketFrame.FrameType.TEXT, ms
    g.content().copy()));
69.             return;
70.         }
71.         if (msg instanceof ContinuationWebSocketFrame) {
72.             out.add(new MyWebSocketFrame(MyWebSocketFrame.FrameType.CONTINUA
    TION, 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] view plaincopy

```java
/**
 * 解码器，将 byte 转成 char
 * @author c.k
 *
 */
public class ByteToCharDecoder extends ByteToMessageDecoder {

    @Override
    protected void decode(ChannelHandlerContext ctx, ByteBuf in, List<Object> out) throws Exception {
        while(in.readableBytes() >= 2){
            out.add(Character.valueOf(in.readChar()));
        }
    }

}
```

[java] view plaincopy

```java
/**
 * 编码器，将 char 转成 byte
 * @author Administrator
 *
 */
public class CharToByteEncoder extends MessageToByteEncoder<Character> {

    @Override
    protected void encode(ChannelHandlerContext ctx, Character msg, ByteBuf out) throws Exception {
        out.writeChar(msg);
    }
}
```

[java] view plaincopy

```java
/**
 * 继承CombinedChannelDuplexHandler，用于绑定解码器和编码器
 * @author c.k
 *
 */
public class CharCodec extends CombinedChannelDuplexHandler<ByteToCharDecoder, CharToByteEncoder> {
    public CharCodec(){
        super(new ByteToCharDecoder(), new CharToByteEncoder());
    }
}
```

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
- ……

第八章：附带的 ChannelHandler 和 Codec

本章介绍

- 使用 SSL/TLS 创建安全的 Netty 程序
- 使用 Netty 创建 HTTP/HTTPS 程序
- 处理空闲连接和超时
- 解码分隔符和基于长度的协议
- 写大数据
- 序列化数据

上一章讲解了如何创建自己的编解码器，我们现在可以用上一章的知识来编写自己的编解码器。不过 Netty 提供了一些标准的 ChannelHandler 和 Codec。Netty 提供了很多协

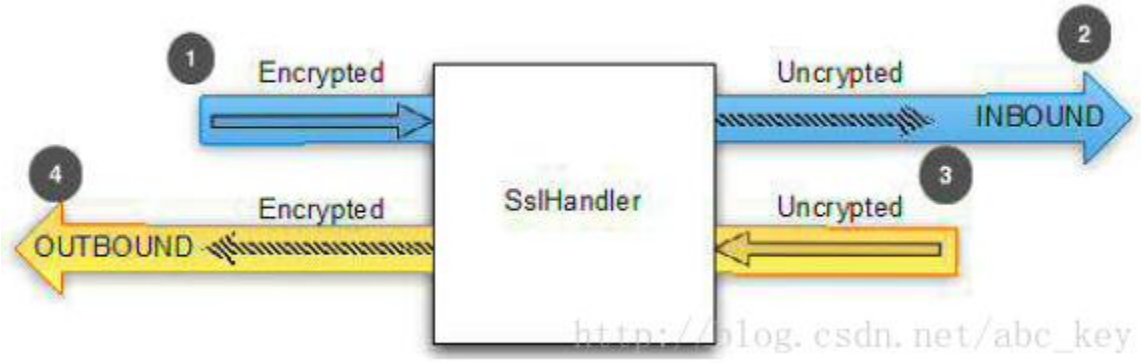
议的支持，所以我们不必自己发明轮子。Netty 提供的这些实现可以解决我们的大部分需求。本章讲解 Netty 中使用 SSL/TLS 编写安全的应用程序，编写 HTTP 协议服务器，以及使用如 WebSocket 或 Google 的 SPDY 协议来使 HTTP 服务获得更好的性能；这些都是很常见的应用，本章还会介绍数据压缩，在数据量比较大的时候，压缩数据是很有必要的。

8.1 使用 SSL/TLS 创建安全的 Netty 程序

通信数据在网络上传输一般是不安全的，因为传输的数据可以发送纯文本或二进制的数据，很容易被破解。我们很有必要对网络上的数据进行加密。SSL 和 TLS 是众所周知的标准和分层的协议，它们可以确保数据时私有的。例如，使用 HTTPS 或 SMTPS 都使用了 SSL/TLS 对数据进行了加密。

对于 SSL/TLS，Java 中提供了抽象的 SslContext 和 SslEngine。实际上，SslContext 可以用来获取 SslEngine 来进行加密和解密。使用指定的加密技术是高度可配置的，但是这不在本章范围。Netty 扩展了 Java 的 SslEngine，添加了一些新功能，使其更适合基于 Netty 的应用程序。Netty 提供的这个扩展是 SslHandler，是 SslEngine 的包装类，用来对网络数据进行加密和解密。

下图显示 SslHandler 实现的数据流：



上图显示了如何使用 ChannelInitializer 将 SslHandler 添加到 ChannelPipeline，看下面代码：

[java] view plaincopy

```
1.  public class SslChannelInitializer extends ChannelInitializer<Channel> {
2.
3.      private final SSLContext context;
4.      private final boolean client;
5.      private final boolean startTls;
6.
7.      public SslChannelInitializer(SSLContext context, boolean client, boolean
    startTls) {
8.          this.context = context;
9.          this.client = client;
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，但是最好这样来做。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，然后选择一个加密密码)，SSL/TLS 是自动执行的。看下面方法列表：

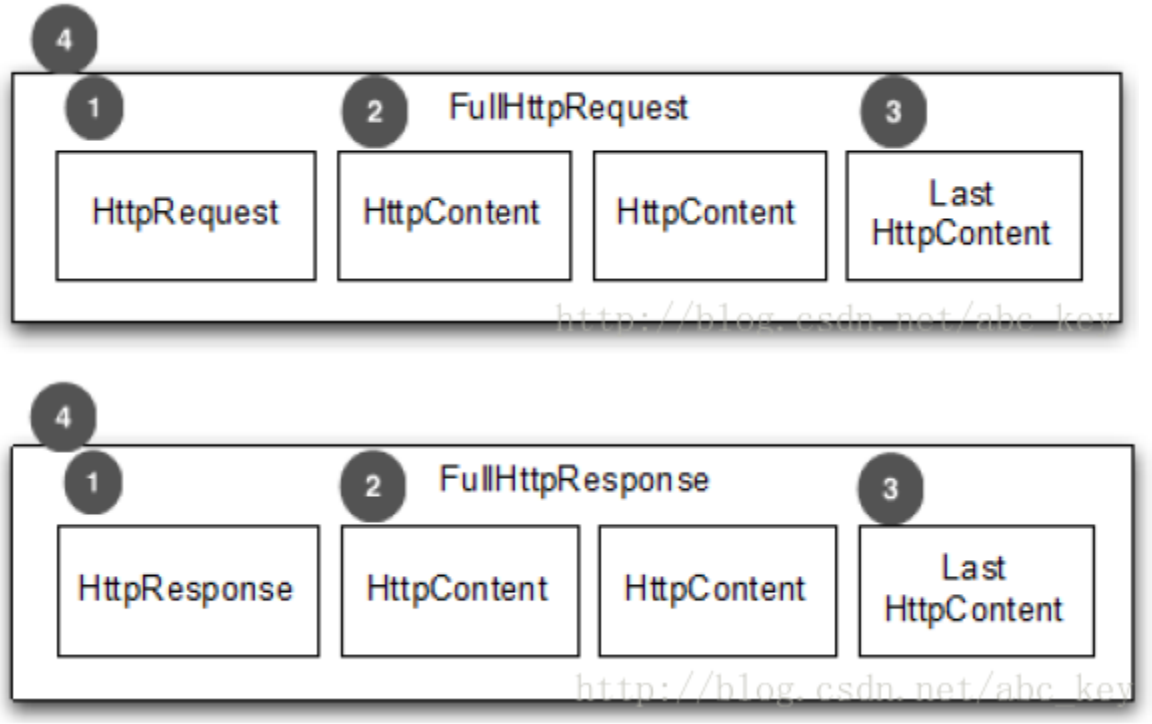
- setHandshakeTimeout(long handshakeTimeout, TimeUnit unit)，设置握手超时时间，ChannelFuture 将得到通知
- setHandshakeTimeoutMillis(long handshakeTimeoutMillis)，设置握手超时时间，ChannelFuture 将得到通知
- getHandshakeTimeoutMillis()，获取握手超时时间值
- setCloseNotifyTimeout(long closeNotifyTimeout, TimeUnit unit)，设置关闭通知超时时间，若超时，ChannelFuture 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

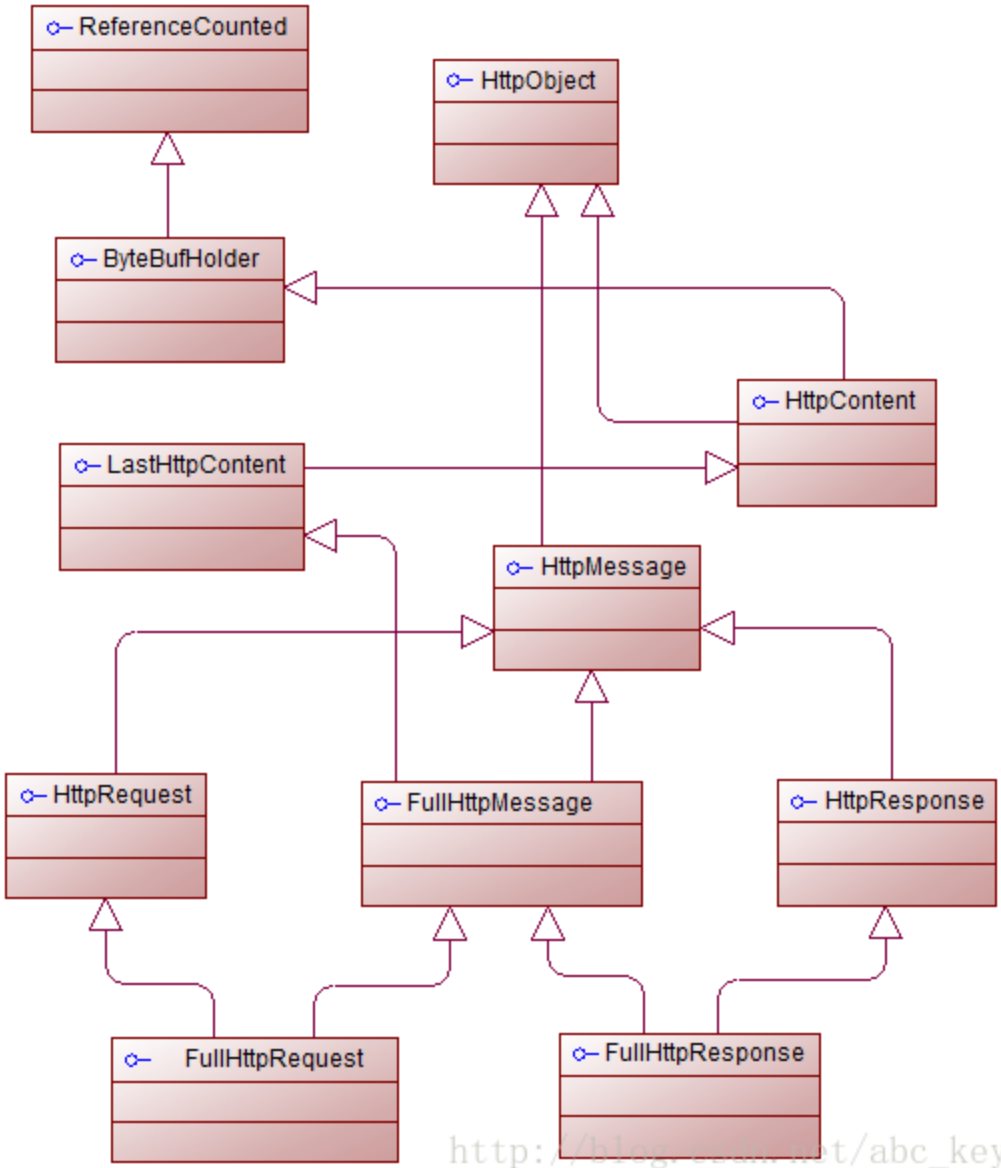
HTTP/HTTPS 是最常用的协议之一，可以通过 HTTP/HTTPS 访问网站，或者是提供对外公开的接口服务等等。Netty 附带了使用 HTTP/HTTPS 的 handlers，而不需要我们自己来编写编解码器。

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] view plaincopy

```
1.  public class HttpDecoderEncoderInitializer extends ChannelInitializer<Channe
    l> {
2.
```

```
3.      private final boolean client;
4.
5.      public HttpDecoderEncoderInitializer(boolean client) {
6.          this.client = client;
7.      }
8.
9.      @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("", 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器简单的编解码器：HttpClientCodec 和 HttpServerCodec。

在 ChannelPipelien 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 请求和响应可以有很多消息数据，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个消息。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] view plaincopy

```
1.  /**
2.   * 添加聚合 http 消息的 Handler
3.   *
4.   * @author c.k
5.   *
6.   */
```

```
7.  public class HttpAggregatorInitializer extends ChannelInitializer<Channel> {
8.
9.      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("aggegator", 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] view plaincopy

```
1.  @Override
2.  protected void initChannel(Channel ch) throws Exception {
3.      ChannelPipeline pipeline = ch.pipeline();
4.      if (client) {
5.          pipeline.addLast("codec", new HttpClientCodec());
6.          //添加解压缩 Handler
7.          pipeline.addLast("decompressor", new HttpContentDecompressor());
8.      } else {
9.          pipeline.addLast("codec", new HttpServerCodec());
10.         //添加解压缩 Handler
11.         pipeline.addLast("decompressor", new HttpContentDecompressor());
12.     }
```

```
13.     pipeline.addLast("aggegator", 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：

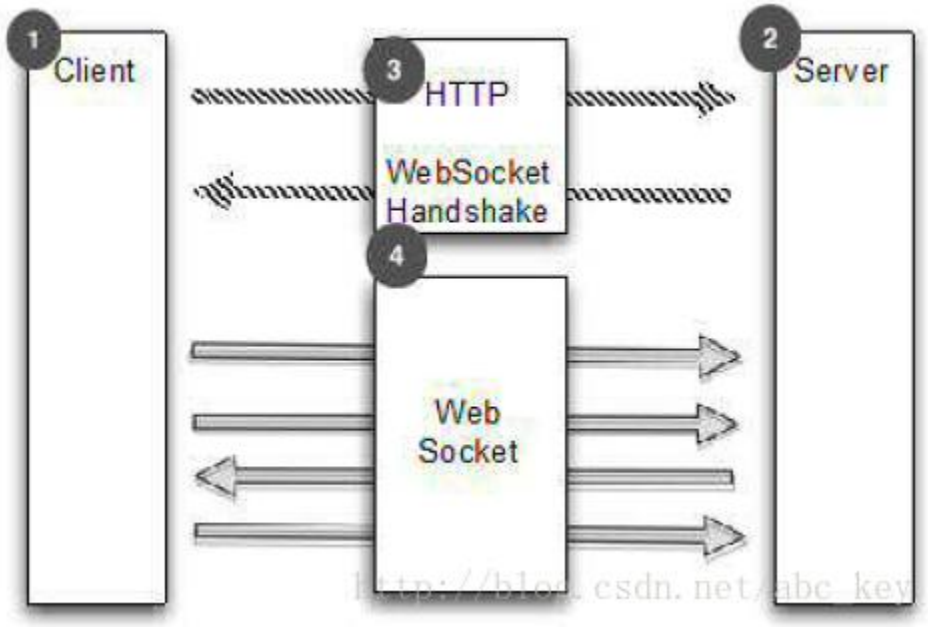
[java] view plaincopy

```
1.  /**
2.   * 使用SSL对HTTP消息加密
3.   *
4.   * @author c.k
5.   *
6.   */
7.  public class HttpsCodecInitializer extends ChannelInitializer<Channel> {
8.
9.      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] view plaincopy

```
1. /**
```

```
 * WebSocket Server，若想使用 SSL 加密，将 SslHandler 加载 ChannelPipeline 的最前面即可
 * @author c.k
 *
 */
public class WebSocketServerInitializer extends ChannelInitializer<Channel> {

    @Override
    protected void initChannel(Channel ch) throws Exception {
        ch.pipeline().addLast(new HttpServerCodec(),
                new HttpObjectAggregator(65536),
                new WebSocketServerProtocolHandler("/websocket"),
                new TextFrameHandler(),
                new BinaryFrameHandler(),
                new ContinuationFrameHandler());
    }

    public static final class TextFrameHandler extends SimpleChannelInboundHandler<TextWebSocketFrame> {
        @Override
        protected void channelRead0(ChannelHandlerContext ctx, TextWebSocketFrame msg) throws Exception {
            // handler text frame
        }
    }

    public static final class BinaryFrameHandler extends SimpleChannelInboundHandler<BinaryWebSocketFrame>{
        @Override
        protected void channelRead0(ChannelHandlerContext ctx, BinaryWebSocketFrame msg) throws Exception {
            //handler binary frame
        }
    }

    public static final class ContinuationFrameHandler extends SimpleChannelInboundHandler<ContinuationWebSocketFrame>{
        @Override
        protected void channelRead0(ChannelHandlerContext ctx, ContinuationWebSocketFrame msg) throws Exception {
            //handler continuation frame
        }
    }
```

```
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] view plaincopy

```java
public class IdleStateHandlerInitializer extends ChannelInitializer<Channel> {

    @Override
    protected void initChannel(Channel ch) throws Exception {
        ChannelPipeline pipeline = ch.pipeline();
        pipeline.addLast(new IdleStateHandler(0, 0, 60, TimeUnit.SECONDS));

        pipeline.addLast(new HeartbeatHandler());
    }

    public static final class HeartbeatHandler extends ChannelInboundHandlerAdapter {
        private static final ByteBuf HEARTBEAT_SEQUENCE = Unpooled.unreleasableBuffer(Unpooled.copiedBuffer(
                "HEARTBEAT", CharsetUtil.UTF_8));

        @Override
        public void userEventTriggered(ChannelHandlerContext ctx, Object evt) throws Exception {
            if (evt instanceof IdleStateEvent) {
                ctx.writeAndFlush(HEARTBEAT_SEQUENCE.duplicate()).addListener(ChannelFutureListener.CLOSE_ON_FAILURE);
            } else {
                super.userEventTriggered(ctx, evt);
            }
        }
    }
}
```

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] view plaincopy

```java
/**
 * 处理换行分隔符消息
 * @author c.k
 *
 */
public class LineBasedHandlerInitializer extends ChannelInitializer<Channel>
 {

    @Override
    protected void initChannel(Channel ch) throws Exception {
        ch.pipeline().addLast(new LineBasedFrameDecoder(65 * 1204), new Fram
eHandler());
    }

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

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] view plaincopy

```java
/**
 * 自定义以分隔符为基础的协议
 * @author c.k
 *
 */
public class CmdHandlerInitializer extends ChannelInitializer<Channel> {

    @Override
    protected void initChannel(Channel ch) throws Exception {
        ch.pipeline().addLast(new CmdDecoder(65 * 1024), new CmdHandler());

    }

    public static final class Cmd {
        private final ByteBuf name;
        private final ByteBuf args;

        public Cmd(ByteBuf name, ByteBuf args) {
            this.name = name;
            this.args = args;
        }

        public ByteBuf getName() {
            return name;
        }

        public ByteBuf getArgs() {
            return args;
        }
    }

    public static final class CmdDecoder extends LineBasedFrameDecoder {

        public CmdDecoder(int maxLength) {
            super(maxLength);
        }

        @Override
        protected Object decode(ChannelHandlerContext ctx, ByteBuf buffer) t
hrows 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.sl
    ice(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) thro
    ws 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：

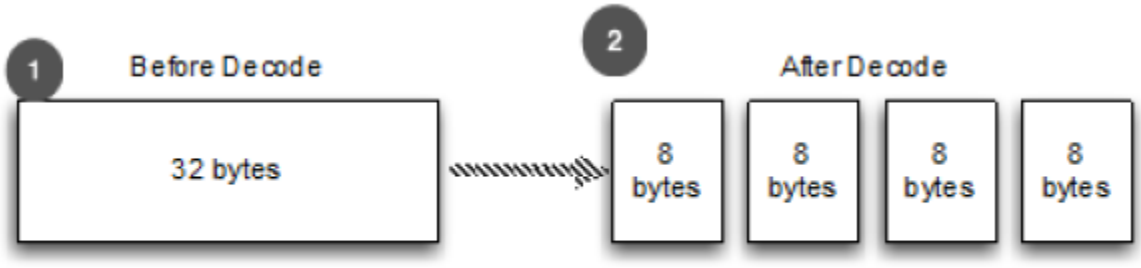


Figure 8.6 Message of fixed size of 8 bytes

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] view plaincopy

```java
public class LengthBasedInitializer extends ChannelInitializer<Channel> {

    @Override
    protected void initChannel(Channel ch) throws Exception {
        ch.pipeline().addLast(new LengthFieldBasedFrameDecoder(65*1024, 0, 8))
            .addLast(new FrameHandler());
    }

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

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] view plaincopy

```java
@Override
public void channelRead(ChannelHandlerContext ctx, Object msg) throws Except
ion {
    File file = new File("test.txt");
    FileInputStream fis = new FileInputStream(file);
    FileRegion region = new DefaultFileRegion(fis.getChannel(), 0, file.leng
th());
    Channel channel = ctx.channel();
    channel.writeAndFlush(region).addListener(new ChannelFutureListener() {

        @Override
        public void operationComplete(ChannelFuture future) throws Exception
 {
            if(!future.isSuccess()){
                Throwable cause = future.cause();
                // do something
            }
        }
    });
}
```

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] view plaincopy

```java
public class ChunkedWriteHandlerInitializer extends ChannelInitializer<Chann
el> {
    private final File file;

    public ChunkedWriteHandlerInitializer(File file) {
        this.file = file;
    }
```

```
7.
8.      @Override
9.      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 ChannelInboundHandlerAdapt
    er {
15.         @Override
16.         public void channelActive(ChannelHandlerContext ctx) throws Exceptio
    n {
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] view plaincopy

```java
/**
 * 使用JBoss Marshalling
 * @author c.k
 *
 */
public class MarshallingInitializer extends ChannelInitializer<Channel> {
    private final MarshallerProvider marshallerProvider;
    private final UnmarshallerProvider unmarshallerProvider;

    public MarshallingInitializer(MarshallerProvider marshallerProvider, Unmarshallerprovider unmarshallerProvider) {
        this.marshallerProvider = marshallerProvider;
        this.unmarshallerProvider = unmarshallerProvider;
    }

    @Override
    protected void initChannel(Channel ch) throws Exception {
        ch.pipeline().addLast(new MarshallingDecoder(unmarshallerProvider))

            .addLast(new MarshallingEncoder(marshallerProvider))
            .addLast(new ObjectHandler());
    }

    public final class ObjectHandler extends SimpleChannelInboundHandler<Serializable> {
        @Override
        protected void channelRead0(ChannelHandlerContext ctx, Serializable msg) throws Exception {
            // do something
        }
    }
```

```
28. }
```

8.6.3 使用 ProtoBuf 序列化

最有一个序列化方案是 Netty 附带的 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 附带的 protobuf 放在 io.netty.handler.codec.protobuf 包下面：

- ProtobufDecoder
- ProtobufEncoder
- ProtobufVarint32FrameDecoder
- ProtobufVarint32LengthFieldPrepender

看下面代码：

[java] view plaincopy

```
1.  /**
2.   * 使用protobuf序列化数据，进行编码解码
3.   * 注意：使用protobuf需要protobuf-java-2.5.0.jar
4.   * @author Administrator
5.   *
6.   */
7.  public class ProtoBufInitializer extends ChannelInitializer<Channel> {
8.
9.      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<Ser
    ializable> {
24.         @Override
```

```
25.         protected void channelRead0(ChannelHandlerContext ctx, Serializable
    msg) throws Exception {
26.             // do something
27.         }
28.     }
29. }
```

第九章：引导 Netty 应用程序

本章介绍

- 引导客户端和服务器
- 从 Channel 引导客户端
- 添加多个 ChannelHandler
- 使用通道选项和属性

上一章学习了编写自己的 ChannelHandler 和编解码器并将它们添加到 Channel 的 ChannelPipeline 中。本章将讲解如何将它们结合在一起使用。

Netty 提供了简单统一的方法来引导服务器和客户端。引导是配置 Netty 服务器和客户端程序的一个过程，Bootstrap 允许这些应用程序很容易的重复使用。Netty 程序的客户端和服务器都可以使用 Bootstrap，其目的是简化编码过程，Bootstrap 还提供了一个机制就是让一些组件(channels,pipeline,handlers 等等)都可以在后台工作。本章将具体结合以下部分一起使用开发 Netty 程序：

- EventLoopGroup
- Channel
- 设置 ChannelOption
- Channel 被注册后将调用 ChannelHandler
- 添加指定的属性到 Channel
- 设置本地和远程地址
- 绑定、连接(取决于类型)

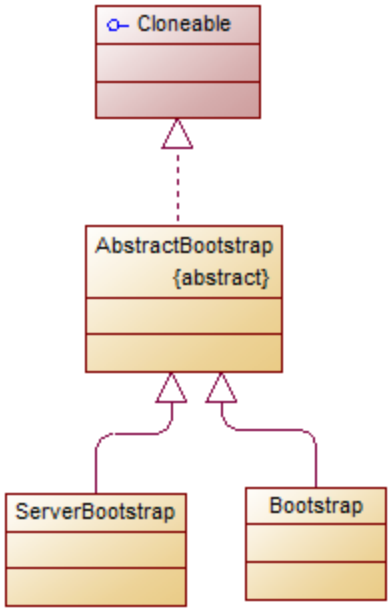
知道如何使用各个 Bootstrap 后就可以使用它们配置服务器和客户端了。本章还将学习在什么会后可以共享一个 Bootstrap 以及为什么这样做，结合我们之前学习的知识点来编写 Netty 程序。

9.1 不同的引导类型

Netty 包含了 2 个不同类型的引导，第一个是使用服务器的 ServerBootstrap，用来接受客户端连接以及为已接受的连接创建子通道；第二个是用于客户端的 Bootstrap，不接受新的连接，并且是在父通道类完成一些操作。

还有一种情况是处理 DatagramChannel 实例，这些用于 UDP 协议，是无连接的。换句话说，由于 UDP 的性质，所以当处理 UDP 数据时没有必要每个连接通道与 TCP 连接一样。因为通道不需要连接后才能发送数据，UDP 是无连接协议。一个通道可以处理所有的数据而不需要依赖子通道。

下图是引导的类关系图：



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器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。

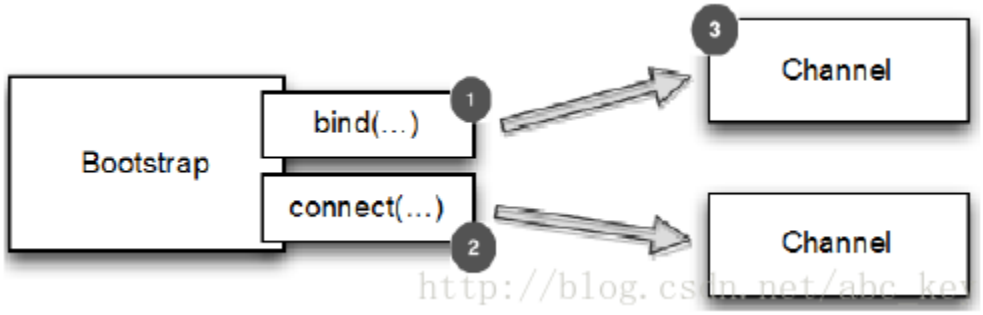
9.2.1 引导客户端的方法

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] view plaincopy

```
1. package netty.in.action;
2.
3. import io.netty.bootstrap.Bootstrap;
4. import io.netty.buffer.ByteBuf;
5. import io.netty.channel.ChannelFuture;
6. import io.netty.channel.ChannelFutureListener;
7. import io.netty.channel.ChannelHandlerContext;
8. import io.netty.channel.EventLoopGroup;
9. import io.netty.channel.SimpleChannelInboundHandler;
```

```
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.nio.NioSocketChannel;

/**
 * 引导配置客户端
 *
 * @author c.k
 *
 */
public class BootstrapingClient {

    public static void main(String[] args) throws Exception {
        EventLoopGroup group = new NioEventLoopGroup();
        Bootstrap b = new Bootstrap();
        b.group(group).channel(NioSocketChannel.class).handler(new SimpleChannelInboundHandler<ByteBuf>() {
            @Override
            protected void channelRead0(ChannelHandlerContext ctx, ByteBuf msg) throws Exception {
                System.out.println("Received data");
                msg.clear();
            }
        });
        ChannelFuture f = b.connect("127.0.0.1", 2048);
        f.addListener(new ChannelFutureListener() {
            @Override
            public void operationComplete(ChannelFuture future) throws Exception {
                if (future.isSuccess()) {
                    System.out.println("connection finished");
                } else {
                    System.out.println("connection failed");
                    future.cause().printStackTrace();
                }
            }
        });
    }
}
```

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

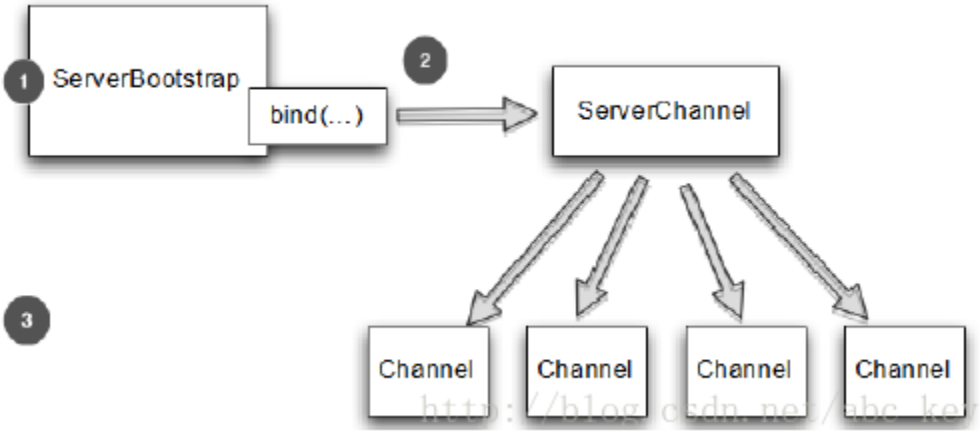
9.3.1 引导服务器的方法

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] view plaincopy

```
1.  package netty.in.action;
2.
3.  import io.netty.bootstrap.ServerBootstrap;
4.  import io.netty.buffer.ByteBuf;
5.  import io.netty.channel.ChannelFuture;
6.  import io.netty.channel.ChannelFutureListener;
7.  import io.netty.channel.ChannelHandlerContext;
8.  import io.netty.channel.EventLoopGroup;
9.  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 BootstrapingServer {
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, B
    yteBuf 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 Excep
    tion {
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. }
```

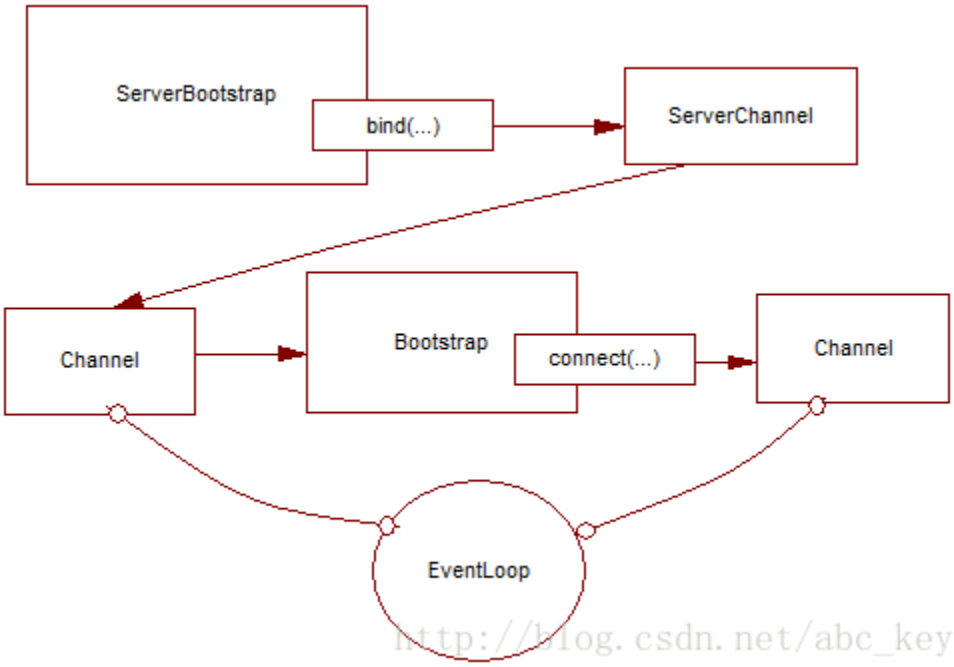
9.4 从 Channel 引导客户端

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 对这方面进行了优化，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] view plaincopy

```java
package netty.in.action;

import java.net.InetSocketAddress;

import io.netty.bootstrap.Bootstrap;
import io.netty.bootstrap.ServerBootstrap;
import io.netty.buffer.ByteBuf;
import io.netty.channel.ChannelFuture;
import io.netty.channel.ChannelFutureListener;
import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.EventLoopGroup;
import io.netty.channel.SimpleChannelInboundHandler;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.nio.NioServerSocketChannel;
import io.netty.channel.socket.nio.NioSocketChannel;

/**
 * 从Channel 引导客户端
 *
 * @author c.k
 *
 */
public class BootstrapingFromChannel {

```

```
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) thr
    ows Exception {
35.                         Bootstrap b = new Bootstrap();
36.                         b.channel(NioSocketChannel.class).handler(
37.                                 new SimpleChannelInboundHandler<ByteBuf>() {

38.                                     @Override
39.                                     protected void channelRead0(ChannelHandl
    erContext 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, B
    yteBuf msg)
51.                             throws Exception {
52.                         if (connectFuture.isDone()) {
53.                             // do something with the data
54.                         }
55.                     }
56.                 });
57.         ChannelFuture f = b.bind(2048);
58.         f.addListener(new ChannelFutureListener() {
59.             @Override
```

```
60.             public void operationComplete(ChannelFuture future) throws Excep
    tion {
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] view plaincopy

```
1.  package netty.in.action;
2.
3.  import io.netty.bootstrap.ServerBootstrap;
4.  import io.netty.channel.Channel;
5.  import io.netty.channel.ChannelFuture;
6.  import io.netty.channel.ChannelInitializer;
7.  import io.netty.channel.EventLoopGroup;
8.  import io.netty.channel.nio.NioEventLoopGroup;
9.  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. /**
```

```
 * 使用ChannelInitializer初始化ChannelHandler
 * @author c.k
 *
 */
public class InitChannelExample {

    public static void main(String[] args) throws Exception {
        EventLoopGroup bossGroup = new NioEventLoopGroup(1);
        EventLoopGroup workerGroup = new NioEventLoopGroup();
        ServerBootstrap b = new ServerBootstrap();
        b.group(bossGroup, workerGroup).channel(NioServerSocketChannel.class
)
            .childHandler(new ChannelInitializerImpl());
        ChannelFuture f = b.bind(2048).sync();
        f.channel().closeFuture().sync();
    }

    static final class ChannelInitializerImpl extends ChannelInitializer<Cha
nnel>{
        @Override
        protected void initChannel(Channel ch) throws Exception {
            ch.pipeline().addLast(new HttpClientCodec())
                .addLast(new HttpObjectAggregator(Integer.MAX_VALUE));
        }
    }

}
```

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] view plaincopy

```java
public static void main(String[] args) {
    //创建属性键对象
    final AttributeKey<Integer> id = AttributeKey.valueOf("ID");
    //客户端引导对象
    Bootstrap b = new Bootstrap();
    //设置 EventLoop，设置通道类型
    b.group(new NioEventLoopGroup()).channel(NioSocketChannel.class)
    //设置 ChannelHandler
        .handler(new SimpleChannelInboundHandler<ByteBuf>() {
            @Override
            protected void channelRead0(ChannelHandlerContext ctx, ByteBuf m
sg)
                    throws Exception {
                System.out.println("Reveived data");
                msg.clear();
            }

            @Override
            public void channelRegistered(ChannelHandlerContext ctx) throws
Exception {
                //通道注册后执行，获取属性值
                Integer idValue = ctx.channel().attr(id).get();
                System.out.println(idValue);
                //do something with the idValue
            }
        });
    //设置通道选项，在通道注册后或被创建后设置
    b.option(ChannelOption.SO_KEEPALIVE, true).option(ChannelOption.CONNECT_
TIMEOUT_MILLIS, 5000);
    //设置通道属性
    b.attr(id, 123456);
    ChannelFuture f = b.connect("www.manning.com",80);
    f.syncUninterruptibly();
}
```

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] view plaincopy

```java
public static void main(String[] args) {
    Bootstrap b = new Bootstrap();
    b.group(new OioEventLoopGroup()).channel(OioDatagramChannel.class)
            .handler(new SimpleChannelInboundHandler<DatagramPacket>() {
                @Override
                protected void channelRead0(ChannelHandlerContext ctx, Datag
ramPacket msg)
                        throws Exception {
                    // do something with the packet
                }
            });
    ChannelFuture f = b.bind(new InetSocketAddress(0));
    f.addListener(new ChannelFutureListener() {
        @Override
        public void operationComplete(ChannelFuture future) throws Exception
 {
            if (future.isSuccess()) {
                System.out.println("Channel bound");
            } else {
                System.err.println("Bound attempt failed");
                future.cause().printStackTrace();
            }
        }
    });
}
```

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!

第十章：单元测试代码

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。

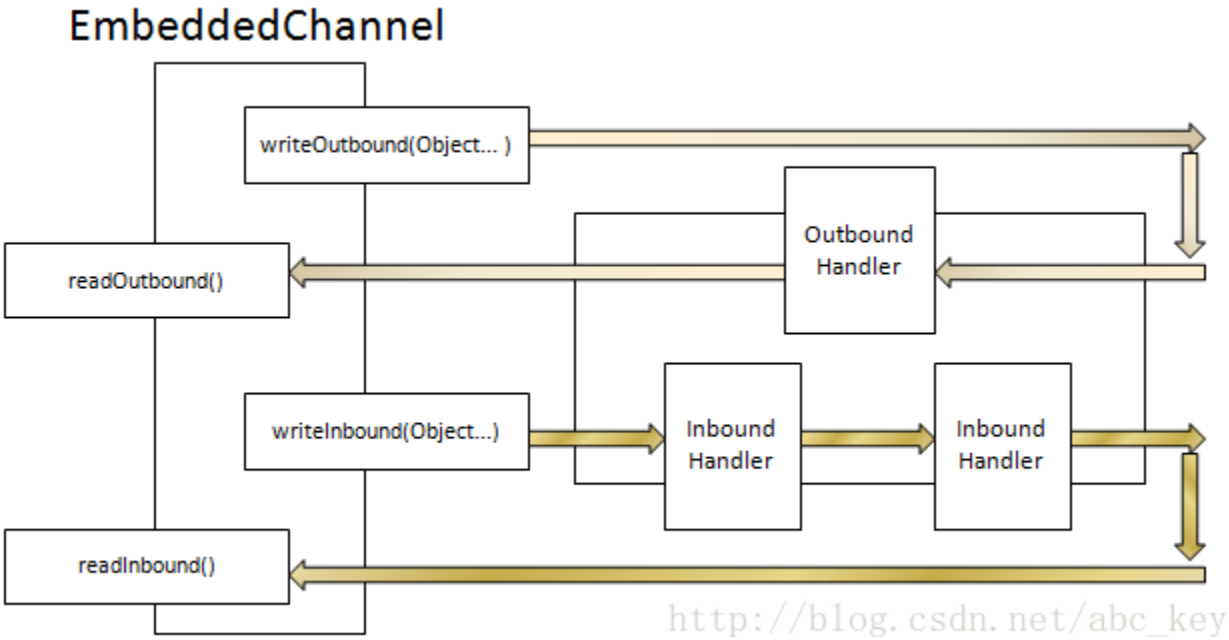
测试 ChannelHandler 可以通过使用“嵌入式”传输很容易的传递事件槽管道以测试你的实现。对于这个嵌入式传输，Netty 提供了一个特定的 Channel 实现：EmbeddedChannel。但是它是如何工作的呢？EmbeddedChannel 的工作非常简单，它允许写入入站或出站数据，然后检查 ChannelPipeline 的结束。这允许你检查消息编码/解码或触发 ChannelHandler 任何行为。

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，有足够的数据可以读取时将产生固定大小的包，如果没有足够的数据可以读取，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] view plaincopy

```
1. package netty.in.action;
2.
3. import java.util.List;
4.
```

```java
import io.netty.buffer.ByteBuf;
import io.netty.channel.ChannelHandlerContext;
import io.netty.handler.codec.ByteToMessageDecoder;

public class FixedLengthFrameDecoder extends ByteToMessageDecoder {

    private final int frameLength;

    public FixedLengthFrameDecoder(int frameLength) {
        if (frameLength <= 0) {
            throw new IllegalArgumentException(
                    "frameLength must be a positive integer: " + frameLength
    );
        }
        this.frameLength = frameLength;
    }

    @Override
    protected void decode(ChannelHandlerContext ctx, ByteBuf in,
            List<Object> out) throws Exception {
        while (in.readableBytes() >= frameLength) {
            ByteBuf buf = in.readBytes(frameLength);
            out.add(buf);
        }
    }

}
```

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] view plaincopy

```java
package netty.in.action;

import io.netty.buffer.ByteBuf;
import io.netty.buffer.Unpooled;
import io.netty.channel.embedded.EmbeddedChannel;

import org.junit.Assert;
import org.junit.Test;

public class FixedLengthFrameDecoderTest {

```

```
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。

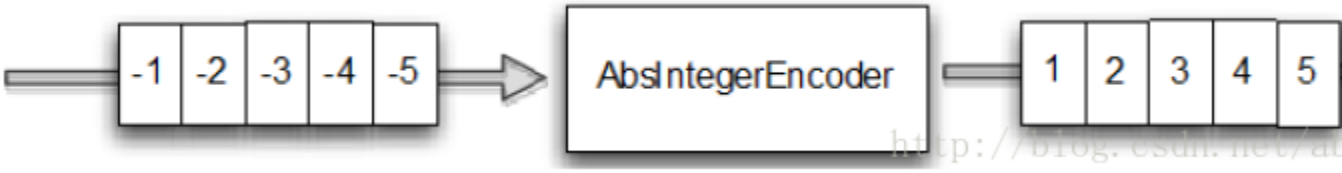
testFramesDecoded2()方法采取同样的方式，但有一个区别就是入站 ByteBuf 分两步写的，当调用 writeInbound(input.readBytes(2))后返回 false 时，FixedLengthFrameDecoder 值会产生输出，至少有 3 个字节是可读，testFramesDecoded2()测试的工作相当于 testFramesDecoded()。

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] view plaincopy

```
1. package netty.in.action;
2.
3. import java.util.List;
4.
5. import io.netty.buffer.ByteBuf;
6. import io.netty.channel.ChannelHandlerContext;
7. import io.netty.handler.codec.MessageToMessageEncoder;
8.
9. 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] view plaincopy

```
1.  package netty.in.action;
2.
3.  import io.netty.buffer.ByteBuf;
4.  import io.netty.buffer.Unpooled;
5.  import io.netty.channel.embedded.EmbeddedChannel;
6.
7.  import org.junit.Assert;
8.  import org.junit.Test;
9.
10. public class AbsIntegerEncoderTest {
11.
12.     @Test
13.     public void testEncoded() {
14.         //创建一个能容纳 10 个 int 的 ByteBuf
15.         ByteBuf 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.         ByteBuf output = (ByteBuf) 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

有时候传输的入站或出站数据不够，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] view plaincopy

```java
package netty.in.action;

import java.util.List;

import io.netty.buffer.ByteBuf;
import io.netty.channel.ChannelHandlerContext;
import io.netty.handler.codec.ByteToMessageDecoder;
import io.netty.handler.codec.TooLongFrameException;

public class FrameChunkDecoder extends ByteToMessageDecoder {

    // 限制大小
    private final int maxFrameSize;

    public FrameChunkDecoder(int maxFrameSize) {
        this.maxFrameSize = maxFrameSize;
    }

    @Override
    protected void decode(ChannelHandlerContext ctx, ByteBuf in,
            List<Object> out) throws Exception {
        // 获取可读字节数
        int readableBytes = in.readableBytes();
        // 若可读字节数大于限制值,清空字节并抛出异常
        if (readableBytes > maxFrameSize) {
            in.clear();
            throw new TooLongFrameException();
        }
        // 读取 ByteBuf 并放到 List 中
        ByteBuf buf = in.readBytes(readableBytes);
        out.add(buf);
```

```
    }

}
```

测试 FrameChunkDecoder 的代码如下：

[java] view plaincopy

```
package netty.in.action;

import io.netty.buffer.ByteBuf;
import io.netty.buffer.Unpooled;
import io.netty.channel.embedded.EmbeddedChannel;
import io.netty.handler.codec.TooLongFrameException;

import org.junit.Assert;
import org.junit.Test;

public class FrameChunkDecoderTest {

    @Test
    public void testFramesDecoded() {
        //创建 ByteBuf 并填充 9 字节数据
        ByteBuf buf = Unpooled.buffer();
        for (int i = 0; i < 9; i++) {
            buf.writeByte(i);
        }
        //复制一个 ByteBuf
        ByteBuf input = buf.duplicate();
        //创建 EmbeddedChannel
        EmbeddedChannel channel = new EmbeddedChannel(new FrameChunkDecoder(
3));
        //读取 2 个字节写入入站通道
        Assert.assertTrue(channel.writeInbound(input.readBytes(2)));
        try {
            //读取 4 个字节写入入站通道
            channel.writeInbound(input.readBytes(4));
            Assert.fail();
        } catch (TooLongFrameException e) {

        }
        //读取 3 个字节写入入站通道
        Assert.assertTrue(channel.writeInbound(input.readBytes(3)));
        //标识完成
        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.

第十一章：WebSocket

本章介绍

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

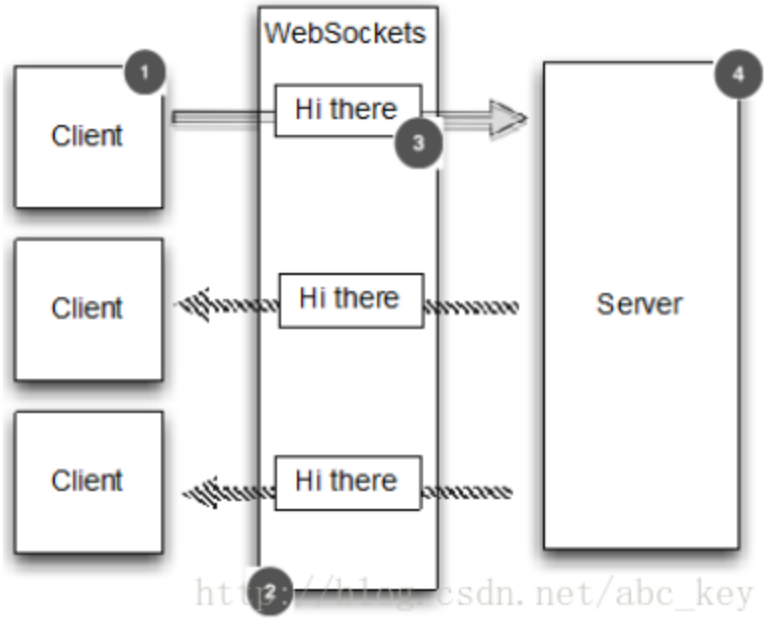
“real-time-web”实时 web 现在随处可见，很多的用户希望能从 web 站点实时获取信息。Netty 支持 WebSocket 实现，并包含了不同的版本，我们可以非常容易的实现 WebSocket 应用。使用 Netty 附带的 WebSocket，我们不需要关注协议内部实现，只需要使用 Netty 提供的一些简单的方法就可以实现。本章将通过的例子应用帮助你来使用 WebSocket 并了解它是如何工作。

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：

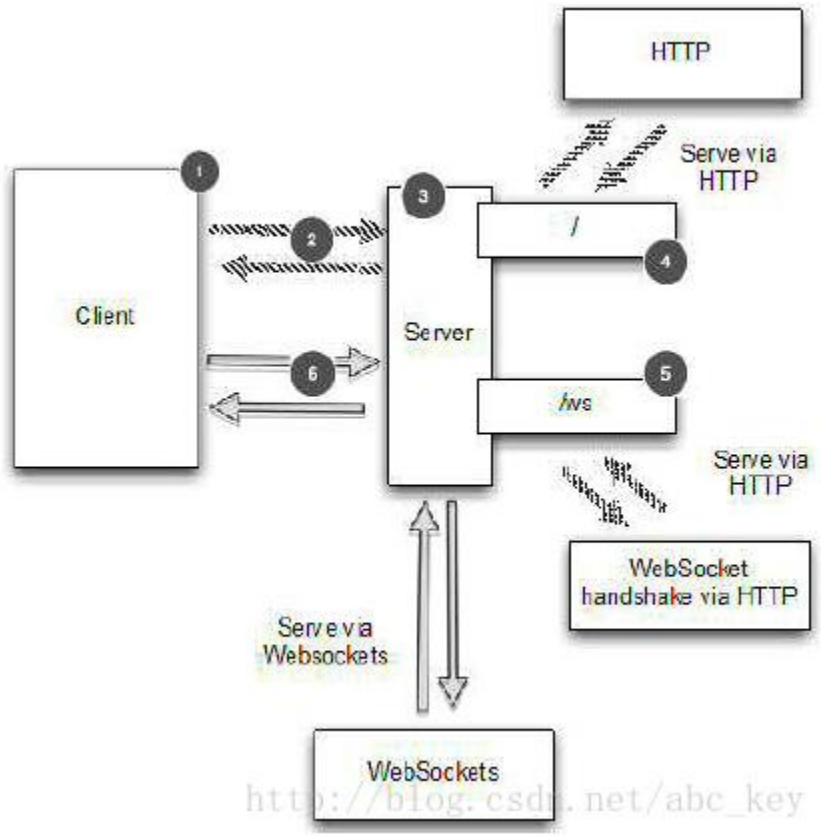
1. 一个客户端发送一条消息
2. 消息被广播到其他已连接的客户端

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] view plaincopy

```
1. package netty.in.action;
2.
3. import io.netty.channel.ChannelFuture;
4. import io.netty.channel.ChannelFutureListener;
5. import io.netty.channel.ChannelHandlerContext;
6. import io.netty.channel.DefaultFileRegion;
7. import io.netty.channel.SimpleChannelInboundHandler;
8. import io.netty.handler.codec.http.DefaultFullHttpResponse;
9. 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 m
    sg)
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(HttpHeaders.Names.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(HttpHeaders.Names.CONTENT_LENGTH,
64.                         file.length());
65.                 response.headers().set(HttpHeaders.Names.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 bina that belongs to a previous BinaryWebSocketFrame or TextWebSocke
CloseWebSocketFrame	WebSocketFrame that represent a CLOSE r and contains close status code and a phrase
PingWebSocketFrame	WebSocketFrame which request the send o PongWebSocketFrame
PongWebSocketFrame	WebSocketFrame which is send as response 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] view plaincopy

```java
package netty.in.action;

import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.SimpleChannelInboundHandler;
import io.netty.channel.group.ChannelGroup;
import io.netty.handler.codec.http.websocketx.TextWebSocketFrame;
import io.netty.handler.codec.http.websocketx.WebSocketServerProtocolHandler
;

/**
 * WebSocket，处理消息
 * @author c.k
 *
 */
```

```
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.chann
    el()
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, TextWebSocketFram
    e 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：

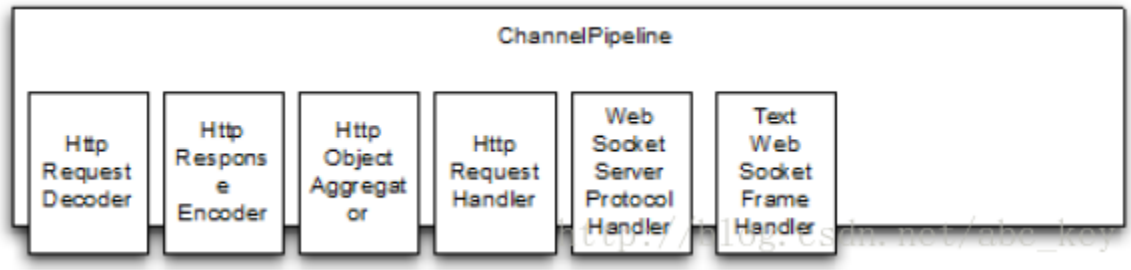
[java] view plaincopy

```
1.  package netty.in.action;
2.
3.  import io.netty.channel.Channel;
```

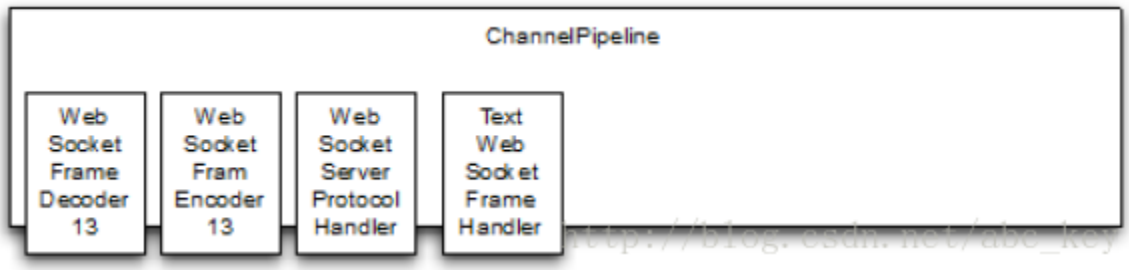
```
4.  import io.netty.channel.ChannelInitializer;
5.  import io.netty.channel.ChannelPipeline;
6.  import io.netty.channel.group.ChannelGroup;
7.  import io.netty.handler.codec.http.HttpObjectAggregator;
8.  import io.netty.handler.codec.http.HttpServerCodec;
9.  import io.netty.handler.codec.http.websocketx.WebSocketServerProtocolHandler
    ;
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 Http 请求的完整性
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. }
```

WebSocketServerProtcolHandler 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] view plaincopy

```
1. package netty.in.action;
2.
3. import io.netty.bootstrap.ServerBootstrap;
4. import io.netty.channel.Channel;
5. import io.netty.channel.ChannelFuture;
6. import io.netty.channel.ChannelInitializer;
7. import io.netty.channel.EventLoopGroup;
8. import io.netty.channel.group.ChannelGroup;
9. 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 gro
    up) {
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();
```

```
            }
        });
        f.channel().closeFuture().syncUninterruptibly();
    }

}
```

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

[html] view plaincopy

```
<html>
<head>
<title>Web Socket Test</title>
</head>
<body>
<script type="text/javascript">
var socket;
if (!window.WebSocket) {
  window.WebSocket = window.MozWebSocket;
}
if (window.WebSocket) {
  socket = new WebSocket("ws://localhost:2048/ws");
  socket.onmessage = function(event) {
    var ta = document.getElementById('responseText');
    ta.value = ta.value + '\n' + event.data
  };
  socket.onopen = function(event) {
    var ta = document.getElementById('responseText');
    ta.value = "Web Socket opened!";
  };
  socket.onclose = function(event) {
    var ta = document.getElementById('responseText');
    ta.value = ta.value + "Web Socket closed";
  };
} else {
  alert("Your browser does not support Web Socket.");
}

function send(message) {
  if (!window.WebSocket) { return; }
  if (socket.readyState == WebSocket.OPEN) {
    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;"></t
    extarea>
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] view plaincopy

```
1.  package netty.in.action;
2.
3.  import io.netty.channel.Channel;
4.  import io.netty.channel.group.ChannelGroup;
5.  import io.netty.handler.ssl.SslHandler;
6.
7.  import javax.net.ssl.SSLContext;
8.  import javax.net.ssl.SSLEngine;
9.
10. public class SecureChatServerIntializer extends ChatServerInitializer {
11.     private final SSLContext context;
12.
13.     public SecureChatServerIntializer(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] view plaincopy

```
1.  package netty.in.action;
2.
3.  import io.netty.channel.Channel;
4.  import io.netty.channel.ChannelFuture;
5.  import io.netty.channel.ChannelInitializer;
6.  import io.netty.channel.group.ChannelGroup;
7.  import java.net.InetSocketAddress;
8.  import javax.net.ssl.SSLContext;
9.
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 gro
    up) {
25.         return new SecureChatServerIntializer(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

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他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 的昵音，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。具体来说是，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：

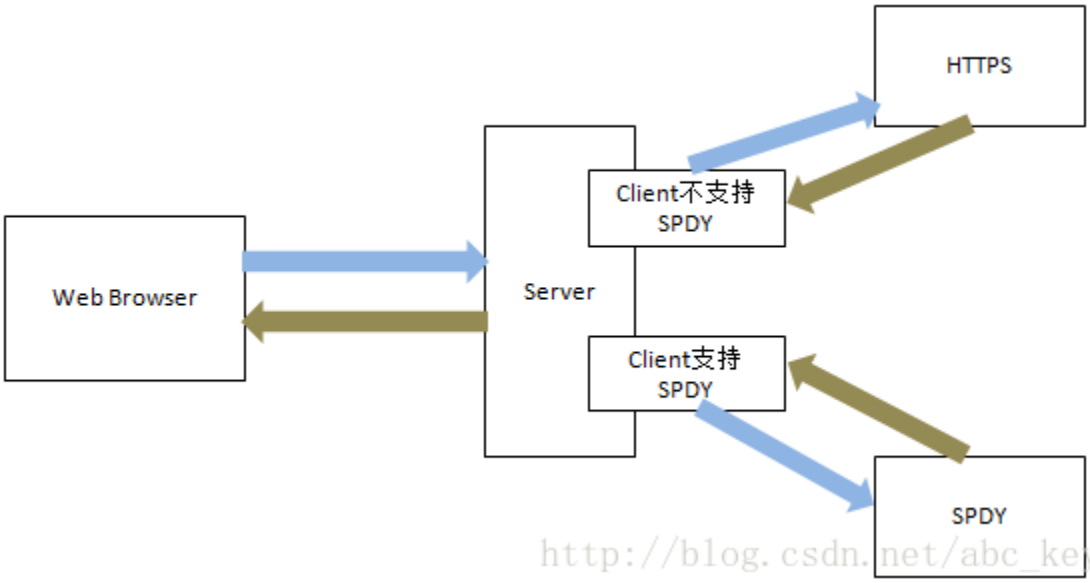
1. SPDY 多路复用，请求优化；而 HTTP 单路连接，请求低效
2. SPDY 支持服务器推送技术；而 HTTP 只允许由客户端主动发起请求
3. SPDY 压缩了 HTTP 头信息，节省了传输数据的带宽流量；而 HTTP 头冗余，同一个会话会反复送头信息
4. SPDY 强制使用 SSL 传输协议，全部请求 SSL 加密后，信息传输更安全

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：

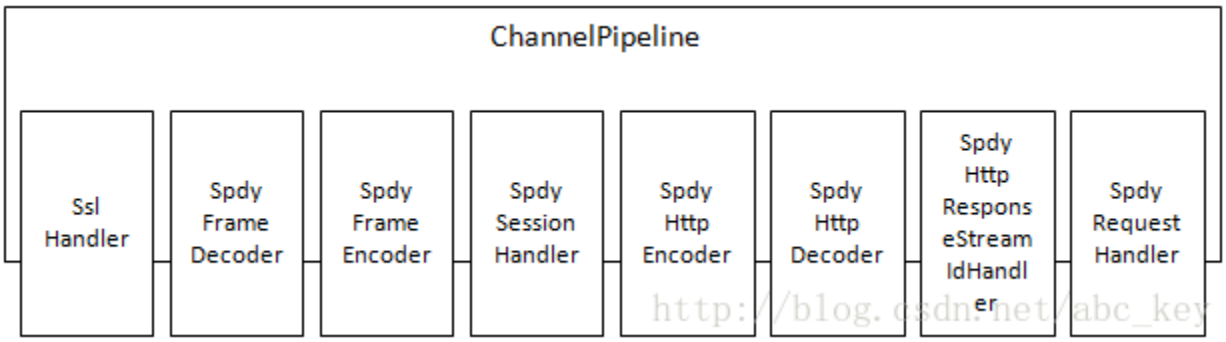
- Google Chrome 19+和 Chromium 19+
- Mozilla Firefox 11+，从 13 开始默认支持
- Opera 12.10+
- Internet Explorer 11+

12.2 本例子流程图

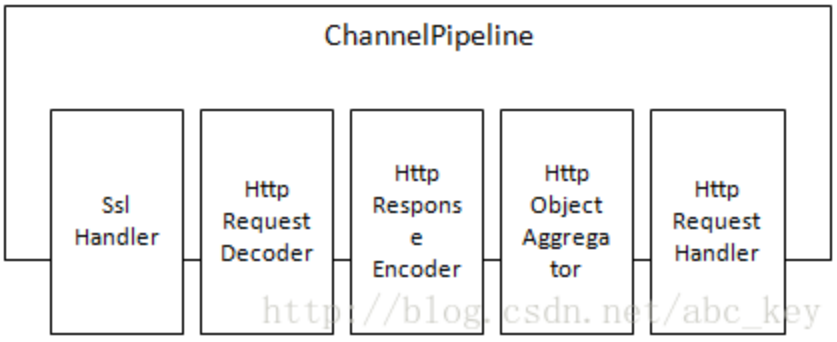


12.3 Netty 中使用 SPDY

支持 SPDY 的 ChannelPipeline 如下图：



不支持 SPDY 的 ChannelPipeline 如下图：



例子代码如下：

[java] view plaincopy

```
1.  package netty.in.action.spdy;
2.
```

```java
import java.util.Arrays;
import java.util.Collections;
import java.util.List;

import org.eclipse.jetty.npn.NextProtoNego.ServerProvider;

public class DefaultServerProvider implements ServerProvider {

    private static final List<String> PROTOCOLS = Collections.unmodifiableList(Arrays
            .asList("spdy/3.1", "http/1.1", "http/1.0", "Unknown"));

    private String protocol;

    public String getSelectedProtocol() {
        return protocol;
    }

    @Override
    public void protocolSelected(String arg0) {
        this.protocol = arg0;
    }

    @Override
    public List<String> protocols() {
        return PROTOCOLS;
    }

    @Override
    public void unsupported() {
        protocol = "http/1.1";
    }

}
```

[java] view plaincopy

```java
package netty.in.action.spdy;

import io.netty.channel.ChannelFuture;
import io.netty.channel.ChannelFutureListener;
import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.SimpleChannelInboundHandler;
import io.netty.handler.codec.http.DefaultFullHttpResponse;
import io.netty.handler.codec.http.FullHttpRequest;
```

```
9.  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<FullHttp
    Request> {
16.
17.     @Override
18.     protected void channelRead0(ChannelHandlerContext ctx, FullHttpRequest r
    equest)
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";
```

```
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)

            throws Exception {
        cause.printStackTrace();
        ctx.close();
    }
}
```

[java] view plaincopy

```
package netty.in.action.spdy;

public class SpdyRequestHandler extends HttpRequestHandler {

    @Override
    protected String getContent() {
        return "This content is transmitted via SPDY\r\n";
    }

}
```

[java] view plaincopy

```
package netty.in.action.spdy;

import io.netty.channel.ChannelInboundHandler;
import io.netty.handler.codec.spdy.SpdyOrHttpChooser;

import javax.net.ssl.SSLEngine;

import org.eclipse.jetty.npn.NextProtoNego;

public class DefaultSpdyOrHttpChooser extends SpdyOrHttpChooser {

    protected DefaultSpdyOrHttpChooser(int maxSpdyContentLength, int maxHttp
ContentLength) {
        super(maxSpdyContentLength, maxHttpContentLength);
    }

    @Override
    protected SelectedProtocol getProtocol(SSLEngine engine) {
```

```
        DefaultServerProvider provider = (DefaultServerProvider) NextProtoNe
go
                .get(engine);
        String protocol = provider.getSelectedProtocol();
        if (protocol == null) {
            return SelectedProtocol.UNKNOWN;
        }
        switch (protocol) {
        case "spdy/3.1":
            return SelectedProtocol.SPDY_3_1;
        case "http/1.0":
        case "http/1.1":
            return SelectedProtocol.HTTP_1_1;
        default:
            return SelectedProtocol.UNKNOWN;
        }
    }

    @Override
    protected ChannelInboundHandler createHttpRequestHandlerForHttp() {
        return new HttpRequestHandler();
    }

    @Override
    protected ChannelInboundHandler createHttpRequestHandlerForSpdy() {
        return new SpdyRequestHandler();
    }

}
```

[java] view plaincopy

```
package netty.in.action.spdy;

import io.netty.channel.Channel;
import io.netty.channel.ChannelInitializer;
import io.netty.channel.ChannelPipeline;
import io.netty.handler.ssl.SslHandler;

import javax.net.ssl.SSLContext;
import javax.net.ssl.SSLEngine;

import org.eclipse.jetty.npn.NextProtoNego;

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 DefaultSpdyOrHttpChooser(1024 * 1024, 1024 * 1024));
30.     }
31. 
32. }
```

[java] view plaincopy

```
1.  package netty.in.action.spdy;
2.  
3.  import io.netty.bootstrap.ServerBootstrap;
4.  import io.netty.channel.Channel;
5.  import io.netty.channel.ChannelFuture;
6.  import io.netty.channel.nio.NioEventLoopGroup;
7.  import io.netty.channel.socket.nio.NioServerSocketChannel;
8.  import io.netty.example.securechat.SecureChatSslContextFactory;
9.  
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] view plaincopy

```
1.  /*
2.   * Copyright 2012 The Netty Project
3.   *
4.   * The Netty Project licenses this file to you under the Apache License,
5.   * version 2.0 (the "License"); you may not use this file except in complian
     ce
6.   * with the License. You may obtain a copy of the License at:
```

```
 *
 *   http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS, WITHOUT

 * WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
 * License for the specific language governing permissions and limitations
 * under the License.
 */
package netty.in.action.spdy;

import javax.net.ssl.ManagerFactoryParameters;
import javax.net.ssl.TrustManager;
import javax.net.ssl.TrustManagerFactorySpi;
import javax.net.ssl.X509TrustManager;
import java.security.InvalidAlgorithmParameterException;
import java.security.KeyStore;
import java.security.KeyStoreException;
import java.security.cert.X509Certificate;

/**
 * Bogus {@link TrustManagerFactorySpi} which accepts any certificate
 * even if it is invalid.
 */
public class SecureChatTrustManagerFactory extends TrustManagerFactorySpi {


    private static final TrustManager DUMMY_TRUST_MANAGER = new X509TrustMan
ager() {
        @Override
        public X509Certificate[] getAcceptedIssuers() {
            return new X509Certificate[0];
        }

        @Override
        public void checkClientTrusted(X509Certificate[] chain, String authT
ype) {
            // Always trust - it is an example.
            // You should do something in the real world.
            // You will reach here only if you enabled client certificate au
th,
            // as described in SecureChatSslContextFactory.
            System.err.println(
```

```
46.                     "UNKNOWN CLIENT CERTIFICATE: " + chain[0].getSubjectDN()
    );
47.         }
48.
49.         @Override
50.         public void checkServerTrusted(X509Certificate[] chain, String authT
    ype) {
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 managerFactoryParamet
    ers)
74.             throws InvalidAlgorithmParameterException {
75.         // Unused
76.     }
77. }
```

[java] view plaincopy

```
1.  /*
2.   * Copyright 2012 The Netty Project
3.   *
4.   * The Netty Project licenses this file to you under the Apache License,
```

```
 * version 2.0 (the "License"); you may not use this file except in compliance
 * with the License. You may obtain a copy of the License at:
 *
 *   http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS, WITHOUT
 * WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
 * License for the specific language governing permissions and limitations
 * under the License.
 */
package netty.in.action.spdy;

import java.io.ByteArrayInputStream;
import java.io.InputStream;

/**
 * A bogus key store which provides all the required information to
 * create an example SSL connection.
 *
 * To generate a bogus key store:
 * <pre>
 * keytool  -genkey -alias securechat -keysize 2048 -validity 36500
 *          -keyalg RSA -dname "CN=securechat"
 *          -keypass secret -storepass secret
 *          -keystore cert.jks
 * </pre>
 */
public final class SecureChatKeyStore {
    private static final short[] DATA = {
        0xfe, 0xed, 0xfe, 0xed, 0x00, 0x00, 0x00, 0x02,
        0x00, 0x00, 0x00, 0x02, 0x00, 0x00, 0x00, 0x01,
        0x00, 0x07, 0x65, 0x78, 0x61, 0x6d, 0x70, 0x6c,
        0x65, 0x00, 0x00, 0x01, 0x1a, 0x9f, 0x57, 0xa5,
        0x27, 0x00, 0x00, 0x01, 0x9a, 0x30, 0x82, 0x01,
        0x96, 0x30, 0x0e, 0x06, 0x0a, 0x2b, 0x06, 0x01,
        0x04, 0x01, 0x2a, 0x02, 0x11, 0x01, 0x01, 0x05,
        0x00, 0x04, 0x82, 0x01, 0x82, 0x48, 0x6d, 0xcf,
        0x16, 0xb5, 0x50, 0x95, 0x36, 0xbf, 0x47, 0x27,
        0x50, 0x58, 0x0d, 0xa2, 0x52, 0x7e, 0x25, 0xab,
        0x14, 0x1a, 0x26, 0x5e, 0x2d, 0x8a, 0x23, 0x90,
        0x60, 0x7f, 0x12, 0x20, 0x56, 0xd1, 0x43, 0xa2,
```

```
        0x6b, 0x47, 0x5d, 0xed, 0x9d, 0xd4, 0xe5, 0x83,
        0x28, 0x89, 0xc2, 0x16, 0x4c, 0x76, 0x06, 0xad,
        0x8e, 0x8c, 0x29, 0x1a, 0x9b, 0x0f, 0xdd, 0x60,
        0x4b, 0xb4, 0x62, 0x82, 0x9e, 0x4a, 0x63, 0x83,
        0x2e, 0xd2, 0x43, 0x78, 0xc2, 0x32, 0x1f, 0x60,
        0xa9, 0x8a, 0x7f, 0x0f, 0x7c, 0xa6, 0x1d, 0xe6,
        0x92, 0x9e, 0x52, 0xc7, 0x7d, 0xbb, 0x35, 0x3b,
        0xaa, 0x89, 0x73, 0x4c, 0xfb, 0x99, 0x54, 0x97,
        0x99, 0x28, 0x6e, 0x66, 0x5b, 0xf7, 0x9b, 0x7e,
        0x6d, 0x8a, 0x2f, 0xfa, 0xc3, 0x1e, 0x71, 0xb9,
        0xbd, 0x8f, 0xc5, 0x63, 0x25, 0x31, 0x20, 0x02,
        0xff, 0x02, 0xf0, 0xc9, 0x2c, 0xdd, 0x3a, 0x10,
        0x30, 0xab, 0xe5, 0xad, 0x3d, 0x1a, 0x82, 0x77,
        0x46, 0xed, 0x03, 0x38, 0xa4, 0x73, 0x6d, 0x36,
        0x36, 0x33, 0x70, 0xb2, 0x63, 0x20, 0xca, 0x03,
        0xbf, 0x5a, 0xf4, 0x7c, 0x35, 0xf0, 0x63, 0x1a,
        0x12, 0x33, 0x12, 0x58, 0xd9, 0xa2, 0x63, 0x6b,
        0x63, 0x82, 0x41, 0x65, 0x70, 0x37, 0x4b, 0x99,
        0x04, 0x9f, 0xdd, 0x5e, 0x07, 0x01, 0x95, 0x9f,
        0x36, 0xe8, 0xc3, 0x66, 0x2a, 0x21, 0x69, 0x68,
        0x40, 0xe6, 0xbc, 0xbb, 0x85, 0x81, 0x21, 0x13,
        0xe6, 0xa4, 0xcf, 0xd3, 0x67, 0xe3, 0xfd, 0x75,
        0xf0, 0xdf, 0x83, 0xe0, 0xc5, 0x36, 0x09, 0xac,
        0x1b, 0xd4, 0xf7, 0x2a, 0x23, 0x57, 0x1c, 0x5c,
        0x0f, 0xf4, 0xcf, 0xa2, 0xcf, 0xf5, 0xbd, 0x9c,
        0x69, 0x98, 0x78, 0x3a, 0x25, 0xe4, 0xfd, 0x85,
        0x11, 0xcc, 0x7d, 0xef, 0xeb, 0x74, 0x60, 0xb1,
        0xb7, 0xfb, 0x1f, 0x0e, 0x62, 0xff, 0xfe, 0x09,
        0x0a, 0xc3, 0x80, 0x2f, 0x10, 0x49, 0x89, 0x78,
        0xd2, 0x08, 0xfa, 0x89, 0x22, 0x45, 0x91, 0x21,
        0xbc, 0x90, 0x3e, 0xad, 0xb3, 0x0a, 0xb4, 0x0e,
        0x1c, 0xa1, 0x93, 0x92, 0xd8, 0x72, 0x07, 0x54,
        0x60, 0xe7, 0x91, 0xfc, 0xd9, 0x3c, 0xe1, 0x6f,
        0x08, 0xe4, 0x56, 0xf6, 0x0b, 0xb0, 0x3c, 0x39,
        0x8a, 0x2d, 0x48, 0x44, 0x28, 0x13, 0xca, 0xe9,
        0xf7, 0xa3, 0xb6, 0x8a, 0x5f, 0x31, 0xa9, 0x72,
        0xf2, 0xde, 0x96, 0xf2, 0xb1, 0x53, 0xb1, 0x3e,
        0x24, 0x57, 0xfd, 0x18, 0x45, 0x1f, 0xc5, 0x33,
        0x1b, 0xa4, 0xe8, 0x21, 0xfa, 0x0e, 0xb2, 0xb9,
        0xcb, 0xc7, 0x07, 0x41, 0xdd, 0x2f, 0xb6, 0x6a,
        0x23, 0x18, 0xed, 0xc1, 0xef, 0xe2, 0x4b, 0xec,
        0xc9, 0xba, 0xfb, 0x46, 0x43, 0x90, 0xd7, 0xb5,
        0x68, 0x28, 0x31, 0x2b, 0x8d, 0xa8, 0x51, 0x63,
        0xf7, 0x53, 0x99, 0x19, 0x68, 0x85, 0x66, 0x00,
```

```
        0x00, 0x00, 0x01, 0x00, 0x05, 0x58, 0x2e, 0x35,
        0x30, 0x39, 0x00, 0x00, 0x02, 0x3a, 0x30, 0x82,
        0x02, 0x36, 0x30, 0x82, 0x01, 0xe0, 0xa0, 0x03,
        0x02, 0x01, 0x02, 0x02, 0x04, 0x48, 0x59, 0xf1,
        0x92, 0x30, 0x0d, 0x06, 0x09, 0x2a, 0x86, 0x48,
        0x86, 0xf7, 0x0d, 0x01, 0x01, 0x05, 0x05, 0x00,
        0x30, 0x81, 0xa0, 0x31, 0x0b, 0x30, 0x09, 0x06,
        0x03, 0x55, 0x04, 0x06, 0x13, 0x02, 0x4b, 0x52,
        0x31, 0x13, 0x30, 0x11, 0x06, 0x03, 0x55, 0x04,
         0x08, 0x13, 0x0a, 0x4b, 0x79, 0x75, 0x6e, 0x67,
         0x67, 0x69, 0x2d, 0x64, 0x6f, 0x31, 0x14, 0x30,
         0x12, 0x06, 0x03, 0x55, 0x04, 0x07, 0x13, 0x0b,
         0x53, 0x65, 0x6f, 0x6e, 0x67, 0x6e, 0x61, 0x6d,
         0x2d, 0x73, 0x69, 0x31, 0x1a, 0x30, 0x18, 0x06,
         0x03, 0x55, 0x04, 0x0a, 0x13, 0x11, 0x54, 0x68,
         0x65, 0x20, 0x4e, 0x65, 0x74, 0x74, 0x79, 0x20,
         0x50, 0x72, 0x6f, 0x6a, 0x65, 0x63, 0x74, 0x31,
         0x18, 0x30, 0x16, 0x06, 0x03, 0x55, 0x04, 0x0b,
         0x13, 0x0f, 0x45, 0x78, 0x61, 0x6d, 0x70, 0x6c,
         0x65, 0x20, 0x41, 0x75, 0x74, 0x68, 0x6f, 0x72,
         0x73, 0x31, 0x30, 0x30, 0x2e, 0x06, 0x03, 0x55,
         0x04, 0x03, 0x13, 0x27, 0x73, 0x65, 0x63, 0x75,
         0x72, 0x65, 0x63, 0x68, 0x61, 0x74, 0x2e, 0x65,
         0x78, 0x61, 0x6d, 0x70, 0x6c, 0x65, 0x2e, 0x6e,
         0x65, 0x74, 0x74, 0x79, 0x2e, 0x67, 0x6c, 0x65,
         0x61, 0x6d, 0x79, 0x6e, 0x6f, 0x64, 0x65, 0x2e,
         0x6e, 0x65, 0x74, 0x30, 0x20, 0x17, 0x0d, 0x30,
         0x38, 0x30, 0x36, 0x31, 0x39, 0x30, 0x35, 0x34,
         0x31, 0x33, 0x38, 0x5a, 0x18, 0x0f, 0x32, 0x31,
         0x38, 0x37, 0x31, 0x31, 0x32, 0x34, 0x30, 0x35,
         0x34, 0x31, 0x33, 0x38, 0x5a, 0x30, 0x81, 0xa0,
         0x31, 0x0b, 0x30, 0x09, 0x06, 0x03, 0x55, 0x04,
         0x06, 0x13, 0x02, 0x4b, 0x52, 0x31, 0x13, 0x30,
         0x11, 0x06, 0x03, 0x55, 0x04, 0x08, 0x13, 0x0a,
         0x4b, 0x79, 0x75, 0x6e, 0x67, 0x67, 0x69, 0x2d,
         0x64, 0x6f, 0x31, 0x14, 0x30, 0x12, 0x06, 0x03,
         0x55, 0x04, 0x07, 0x13, 0x0b, 0x53, 0x65, 0x6f,
         0x6e, 0x67, 0x6e, 0x61, 0x6d, 0x2d, 0x73, 0x69,
         0x31, 0x1a, 0x30, 0x18, 0x06, 0x03, 0x55, 0x04,
         0x0a, 0x13, 0x11, 0x54, 0x68, 0x65, 0x20, 0x4e,
         0x65, 0x74, 0x74, 0x79, 0x20, 0x50, 0x72, 0x6f,
         0x6a, 0x65, 0x63, 0x74, 0x31, 0x18, 0x30, 0x16,
         0x06, 0x03, 0x55, 0x04, 0x0b, 0x13, 0x0f, 0x45,
         0x78, 0x61, 0x6d, 0x70, 0x6c, 0x65, 0x20, 0x41,
```

```
        0x75, 0x74, 0x68, 0x6f, 0x72, 0x73, 0x31, 0x30,
        0x30, 0x2e, 0x06, 0x03, 0x55, 0x04, 0x03, 0x13,
        0x27, 0x73, 0x65, 0x63, 0x75, 0x72, 0x65, 0x63,
        0x68, 0x61, 0x74, 0x2e, 0x65, 0x78, 0x61, 0x6d,
        0x70, 0x6c, 0x65, 0x2e, 0x6e, 0x65, 0x74, 0x74,
        0x79, 0x2e, 0x67, 0x6c, 0x65, 0x61, 0x6d, 0x79,
        0x6e, 0x6f, 0x64, 0x65, 0x2e, 0x6e, 0x65, 0x74,
        0x30, 0x5c, 0x30, 0x0d, 0x06, 0x09, 0x2a, 0x86,
        0x48, 0x86, 0xf7, 0x0d, 0x01, 0x01, 0x01, 0x05,
        0x00, 0x03, 0x4b, 0x00, 0x30, 0x48, 0x02, 0x41,
        0x00, 0xc3, 0xe3, 0x5e, 0x41, 0xa7, 0x87, 0x11,
        0x00, 0x42, 0x2a, 0xb0, 0x4b, 0xed, 0xb2, 0xe0,
        0x23, 0xdb, 0xb1, 0x3d, 0x58, 0x97, 0x35, 0x60,
        0x0b, 0x82, 0x59, 0xd3, 0x00, 0xea, 0xd4, 0x61,
        0xb8, 0x79, 0x3f, 0xb6, 0x3c, 0x12, 0x05, 0x93,
        0x2e, 0x9a, 0x59, 0x68, 0x14, 0x77, 0x3a, 0xc8,
        0x50, 0x25, 0x57, 0xa4, 0x49, 0x18, 0x63, 0x41,
        0xf0, 0x2d, 0x28, 0xec, 0x06, 0xfb, 0xb4, 0x9f,
        0xbf, 0x02, 0x03, 0x01, 0x00, 0x01, 0x30, 0x0d,
        0x06, 0x09, 0x2a, 0x86, 0x48, 0x86, 0xf7, 0x0d,
        0x01, 0x01, 0x05, 0x05, 0x00, 0x03, 0x41, 0x00,
        0x65, 0x6c, 0x30, 0x01, 0xc2, 0x8e, 0x3e, 0xcb,
        0xb3, 0x77, 0x48, 0xe9, 0x66, 0x61, 0x9a, 0x40,
        0x86, 0xaf, 0xf6, 0x03, 0xeb, 0xba, 0x6a, 0xf2,
        0xfd, 0xe2, 0xaf, 0x36, 0x5e, 0x7b, 0xaa, 0x22,
        0x04, 0xdd, 0x2c, 0x20, 0xc4, 0xfc, 0xdd, 0xd0,
        0x82, 0x20, 0x1c, 0x3d, 0xd7, 0x9e, 0x5e, 0x5c,
        0x92, 0x5a, 0x76, 0x71, 0x28, 0xf5, 0x07, 0x7d,
        0xa2, 0x81, 0xba, 0x77, 0x9f, 0x2a, 0xd9, 0x44,
        0x00, 0x00, 0x00, 0x01, 0x00, 0x05, 0x6d, 0x79,
        0x6b, 0x65, 0x79, 0x00, 0x00, 0x01, 0x1a, 0x9f,
        0x5b, 0x56, 0xa0, 0x00, 0x00, 0x01, 0x99, 0x30,
        0x82, 0x01, 0x95, 0x30, 0x0e, 0x06, 0x0a, 0x2b,
        0x06, 0x01, 0x04, 0x01, 0x2a, 0x02, 0x11, 0x01,
        0x01, 0x05, 0x00, 0x04, 0x82, 0x01, 0x81, 0x29,
        0xa8, 0xb6, 0x08, 0x0c, 0x85, 0x75, 0x3e, 0xdd,
        0xb5, 0xe5, 0x1a, 0x87, 0x68, 0xd1, 0x90, 0x4b,
        0x29, 0x31, 0xee, 0x90, 0xbc, 0x9d, 0x73, 0xa0,
        0x3f, 0xe9, 0x0b, 0xa4, 0xef, 0x30, 0x9b, 0x36,
        0x9a, 0xb2, 0x54, 0x77, 0x81, 0x07, 0x4b, 0xaa,
        0xa5, 0x77, 0x98, 0xe1, 0xeb, 0xb5, 0x7c, 0x4e,
        0x48, 0xd5, 0x08, 0xfc, 0x2c, 0x36, 0xe2, 0x65,
        0x03, 0xac, 0xe5, 0xf3, 0x96, 0xb7, 0xd0, 0xb5,
        0x3b, 0x92, 0xe4, 0x14, 0x05, 0x7a, 0x6a, 0x92,
```

```
        0x56, 0xfe, 0x4e, 0xab, 0xd3, 0x0e, 0x32, 0x04,
        0x22, 0x22, 0x74, 0x47, 0x7d, 0xec, 0x21, 0x99,
        0x30, 0x31, 0x64, 0x46, 0x64, 0x9b, 0xc7, 0x13,
        0xbf, 0xbe, 0xd0, 0x31, 0x49, 0xe7, 0x3c, 0xbf,
        0xba, 0xb1, 0x20, 0xf9, 0x42, 0xf4, 0xa9, 0xa9,
        0xe5, 0x13, 0x65, 0x32, 0xbf, 0x7c, 0xcc, 0x91,
        0xd3, 0xfd, 0x24, 0x47, 0x0b, 0xe5, 0x53, 0xad,
        0x50, 0x30, 0x56, 0xd1, 0xfa, 0x9c, 0x37, 0xa8,
        0xc1, 0xce, 0xf6, 0x0b, 0x18, 0xaa, 0x7c, 0xab,
        0xbd, 0x1f, 0xdf, 0xe4, 0x80, 0xb8, 0xa7, 0xe0,
        0xad, 0x7d, 0x50, 0x74, 0xf1, 0x98, 0x78, 0xbc,
        0x58, 0xb9, 0xc2, 0x52, 0xbe, 0xd2, 0x5b, 0x81,
        0x94, 0x83, 0x8f, 0xb9, 0x4c, 0xee, 0x01, 0x2b,
        0x5e, 0xc9, 0x6e, 0x9b, 0xf5, 0x63, 0x69, 0xe4,
        0xd8, 0x0b, 0x47, 0xd8, 0xfd, 0xd8, 0xe0, 0xed,
        0xa8, 0x27, 0x03, 0x74, 0x1e, 0x5d, 0x32, 0xe6,
        0x5c, 0x63, 0xc2, 0xfb, 0x3f, 0xee, 0xb4, 0x13,
        0xc6, 0x0e, 0x6e, 0x74, 0xe0, 0x22, 0xac, 0xce,
        0x79, 0xf9, 0x43, 0x68, 0xc1, 0x03, 0x74, 0x2b,
        0xe1, 0x18, 0xf8, 0x7f, 0x76, 0x9a, 0xea, 0x82,
        0x3f, 0xc2, 0xa6, 0xa7, 0x4c, 0xfe, 0xae, 0x29,
        0x3b, 0xc1, 0x10, 0x7c, 0xd5, 0x77, 0x17, 0x79,
        0x5f, 0xcb, 0xad, 0x1f, 0xd8, 0xa1, 0xfd, 0x90,
        0xe1, 0x6b, 0xb2, 0xef, 0xb9, 0x41, 0x26, 0xa4,
        0x0b, 0x4f, 0xc6, 0x83, 0x05, 0x6f, 0xf0, 0x64,
        0x40, 0xe1, 0x44, 0xc4, 0xf9, 0x40, 0x2b, 0x3b,
        0x40, 0xdb, 0xaf, 0x35, 0xa4, 0x9b, 0x9f, 0xc4,
        0x74, 0x07, 0xe5, 0x18, 0x60, 0xc5, 0xfe, 0x15,
        0x0e, 0x3a, 0x25, 0x2a, 0x11, 0xee, 0x78, 0x2f,
        0xb8, 0xd1, 0x6e, 0x4e, 0x3c, 0x0a, 0xb5, 0xb9,
        0x40, 0x86, 0x27, 0x6d, 0x8f, 0x53, 0xb7, 0x77,
        0x36, 0xec, 0x5d, 0xed, 0x32, 0x40, 0x43, 0x82,
        0xc3, 0x52, 0x58, 0xc4, 0x26, 0x39, 0xf3, 0xb3,
        0xad, 0x58, 0xab, 0xb7, 0xf7, 0x8e, 0x0e, 0xba,
        0x8e, 0x78, 0x9d, 0xbf, 0x58, 0x34, 0xbd, 0x77,
        0x73, 0xa6, 0x50, 0x55, 0x00, 0x60, 0x26, 0xbf,
        0x6d, 0xb4, 0x98, 0x8a, 0x18, 0x83, 0x89, 0xf8,
        0xcd, 0x0d, 0x49, 0x06, 0xae, 0x51, 0x6e, 0xaf,
        0xbd, 0xe2, 0x07, 0x13, 0xd8, 0x64, 0xcc, 0xbf,
        0x00, 0x00, 0x00, 0x01, 0x00, 0x05, 0x58, 0x2e,
        0x35, 0x30, 0x39, 0x00, 0x00, 0x02, 0x34, 0x30,
        0x82, 0x02, 0x30, 0x30, 0x82, 0x01, 0xda, 0xa0,
        0x03, 0x02, 0x01, 0x02, 0x02, 0x04, 0x48, 0x59,
        0xf2, 0x84, 0x30, 0x0d, 0x06, 0x09, 0x2a, 0x86,
```

```
        0x48, 0x86, 0xf7, 0x0d, 0x01, 0x01, 0x05, 0x05,
        0x00, 0x30, 0x81, 0x9d, 0x31, 0x0b, 0x30, 0x09,
        0x06, 0x03, 0x55, 0x04, 0x06, 0x13, 0x02, 0x4b,
        0x52, 0x31, 0x13, 0x30, 0x11, 0x06, 0x03, 0x55,
        0x04, 0x08, 0x13, 0x0a, 0x4b, 0x79, 0x75, 0x6e,
        0x67, 0x67, 0x69, 0x2d, 0x64, 0x6f, 0x31, 0x14,
        0x30, 0x12, 0x06, 0x03, 0x55, 0x04, 0x07, 0x13,
        0x0b, 0x53, 0x65, 0x6f, 0x6e, 0x67, 0x6e, 0x61,
        0x6d, 0x2d, 0x73, 0x69, 0x31, 0x1a, 0x30, 0x18,
        0x06, 0x03, 0x55, 0x04, 0x0a, 0x13, 0x11, 0x54,
        0x68, 0x65, 0x20, 0x4e, 0x65, 0x74, 0x74, 0x79,
        0x20, 0x50, 0x72, 0x6f, 0x6a, 0x65, 0x63, 0x74,
        0x31, 0x15, 0x30, 0x13, 0x06, 0x03, 0x55, 0x04,
        0x0b, 0x13, 0x0c, 0x43, 0x6f, 0x6e, 0x74, 0x72,
        0x69, 0x62, 0x75, 0x74, 0x6f, 0x72, 0x73, 0x31,
        0x30, 0x30, 0x2e, 0x06, 0x03, 0x55, 0x04, 0x03,
        0x13, 0x27, 0x73, 0x65, 0x63, 0x75, 0x72, 0x65,
        0x63, 0x68, 0x61, 0x74, 0x2e, 0x65, 0x78, 0x61,
        0x6d, 0x70, 0x6c, 0x65, 0x2e, 0x6e, 0x65, 0x74,
        0x74, 0x79, 0x2e, 0x67, 0x6c, 0x65, 0x61, 0x6d,
        0x79, 0x6e, 0x6f, 0x64, 0x65, 0x2e, 0x6e, 0x65,
        0x74, 0x30, 0x20, 0x17, 0x0d, 0x30, 0x38, 0x30,
        0x36, 0x31, 0x39, 0x30, 0x35, 0x34, 0x35, 0x34,
        0x30, 0x5a, 0x18, 0x0f, 0x32, 0x31, 0x38, 0x37,
        0x31, 0x31, 0x32, 0x33, 0x30, 0x35, 0x34, 0x35,
        0x34, 0x30, 0x5a, 0x30, 0x81, 0x9d, 0x31, 0x0b,
        0x30, 0x09, 0x06, 0x03, 0x55, 0x04, 0x06, 0x13,
        0x02, 0x4b, 0x52, 0x31, 0x13, 0x30, 0x11, 0x06,
        0x03, 0x55, 0x04, 0x08, 0x13, 0x0a, 0x4b, 0x79,
        0x75, 0x6e, 0x67, 0x67, 0x69, 0x2d, 0x64, 0x6f,
        0x31, 0x14, 0x30, 0x12, 0x06, 0x03, 0x55, 0x04,
        0x07, 0x13, 0x0b, 0x53, 0x65, 0x6f, 0x6e, 0x67,
        0x6e, 0x61, 0x6d, 0x2d, 0x73, 0x69, 0x31, 0x1a,
        0x30, 0x18, 0x06, 0x03, 0x55, 0x04, 0x0a, 0x13,
        0x11, 0x54, 0x68, 0x65, 0x20, 0x4e, 0x65, 0x74,
        0x74, 0x79, 0x20, 0x50, 0x72, 0x6f, 0x6a, 0x65,
        0x63, 0x74, 0x31, 0x15, 0x30, 0x13, 0x06, 0x03,
        0x55, 0x04, 0x0b, 0x13, 0x0c, 0x43, 0x6f, 0x6e,
        0x74, 0x72, 0x69, 0x62, 0x75, 0x74, 0x6f, 0x72,
        0x73, 0x31, 0x30, 0x30, 0x2e, 0x06, 0x03, 0x55,
        0x04, 0x03, 0x13, 0x27, 0x73, 0x65, 0x63, 0x75,
        0x72, 0x65, 0x63, 0x68, 0x61, 0x74, 0x2e, 0x65,
        0x78, 0x61, 0x6d, 0x70, 0x6c, 0x65, 0x2e, 0x6e,
        0x65, 0x74, 0x74, 0x79, 0x2e, 0x67, 0x6c, 0x65,
```

```
        0x61, 0x6d, 0x79, 0x6e, 0x6f, 0x64, 0x65, 0x2e,
        0x6e, 0x65, 0x74, 0x30, 0x5c, 0x30, 0x0d, 0x06,
        0x09, 0x2a, 0x86, 0x48, 0x86, 0xf7, 0x0d, 0x01,
        0x01, 0x01, 0x05, 0x00, 0x03, 0x4b, 0x00, 0x30,
        0x48, 0x02, 0x41, 0x00, 0x95, 0xb3, 0x47, 0x17,
        0x95, 0x0f, 0x57, 0xcf, 0x66, 0x72, 0x0a, 0x7e,
        0x5b, 0x54, 0xea, 0x8c, 0x6f, 0x79, 0xde, 0x94,
        0xac, 0x0b, 0x5a, 0xd4, 0xd6, 0x1b, 0x58, 0x12,
        0x1a, 0x16, 0x3d, 0xfe, 0xdf, 0xa5, 0x2b, 0x86,
        0xbc, 0x64, 0xd4, 0x80, 0x1e, 0x3f, 0xf9, 0xe2,
        0x04, 0x03, 0x79, 0x9b, 0xc1, 0x5c, 0xf0, 0xf1,
        0xf3, 0xf1, 0xe3, 0xbf, 0x3f, 0xc0, 0x1f, 0xdd,
        0xdb, 0xc0, 0x5b, 0x21, 0x02, 0x03, 0x01, 0x00,
        0x01, 0x30, 0x0d, 0x06, 0x09, 0x2a, 0x86, 0x48,
        0x86, 0xf7, 0x0d, 0x01, 0x01, 0x05, 0x05, 0x00,
        0x03, 0x41, 0x00, 0x02, 0xd7, 0xdd, 0xbd, 0x0c,
        0x8e, 0x21, 0x20, 0xef, 0x9e, 0x4f, 0x1f, 0xf5,
        0x49, 0xf1, 0xae, 0x58, 0x9b, 0x94, 0x3a, 0x1f,
        0x70, 0x33, 0xf0, 0x9b, 0xbb, 0xe9, 0xc0, 0xf3,
        0x72, 0xcb, 0xde, 0xb6, 0x56, 0x72, 0xcc, 0x1c,
        0xf0, 0xd6, 0x5a, 0x2a, 0xbc, 0xa1, 0x7e, 0x23,
        0x83, 0xe9, 0xe7, 0xcf, 0x9e, 0xa5, 0xf9, 0xcc,
        0xc2, 0x61, 0xf4, 0xdb, 0x40, 0x93, 0x1d, 0x63,
        0x8a, 0x50, 0x4c, 0x11, 0x39, 0xb1, 0x91, 0xc1,
        0xe6, 0x9d, 0xd9, 0x1a, 0x62, 0x1b, 0xb8, 0xd3,
        0xd6, 0x9a, 0x6d, 0xb9, 0x8e, 0x15, 0x51 };

    public static InputStream asInputStream() {
        byte[] data = new byte[DATA.length];
        for (int i = 0; i < data.length; i ++) {
            data[i] = (byte) DATA[i];
        }
        return new ByteArrayInputStream(data);
    }

    public static char[] getCertificatePassword() {
        return "secret".toCharArray();
    }

    public static char[] getKeyStorePassword() {
        return "secret".toCharArray();
    }

    private SecureChatKeyStore() {
```

```
311.         // Unused
312.     }
313. }
```

[java] view plaincopy

```
1.  /*
2.   * Copyright 2012 The Netty Project
3.   *
4.   * The Netty Project licenses this file to you under the Apache License,
5.   * version 2.0 (the "License"); you may not use this file except in complian
     ce
6.   * with the License. You may obtain a copy of the License at:
7.   *
8.   *   http://www.apache.org/licenses/LICENSE-2.0
9.   *
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 thi
     s
32.  * factory accepts any certificate even if it is invalid.  A server-side con
     text
33.  * created by this factory sends a bogus certificate defined in {@link Secur
     eChatKeyStore}.
34.  * <p>
```

```
35.  * You will have to create your context differently in a real world applicat
     ion.
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 a
     s
46.  *     the first argument of {@link SSLContext#init(KeyManager[], TrustManag
     er[], 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.alg
     orithm");
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 kmf = KeyManagerFactory.getInstance(algorithm)
     ;
```

```
74.             kmf.init(ks, SecureChatKeyStore.getCertificatePassword());
75.
76.             // Initialize the SSLContext to work with our key managers.
77.             serverContext = SSLContext.getInstance(PROTOCOL);
78.             serverContext.init(kmf.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.getTrustM
    anagers(), 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

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器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环境下，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

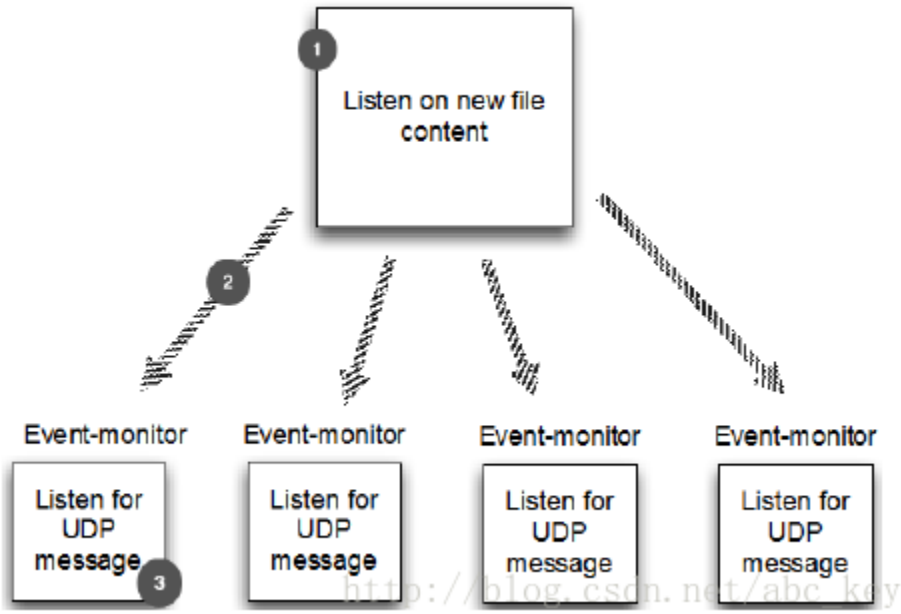
本章例子中，程序打开一个文件并将文件内容一行一行的通过 UDP 广播到其他的接收主机，这很像 UNIX 操作系统的日志系统。对于像发送日志的需求，UDP 非常适合这样的应用程序，并可以使用 UDP 通过网络发送大量的“事件”。

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，因此一台服务机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] view plaincopy

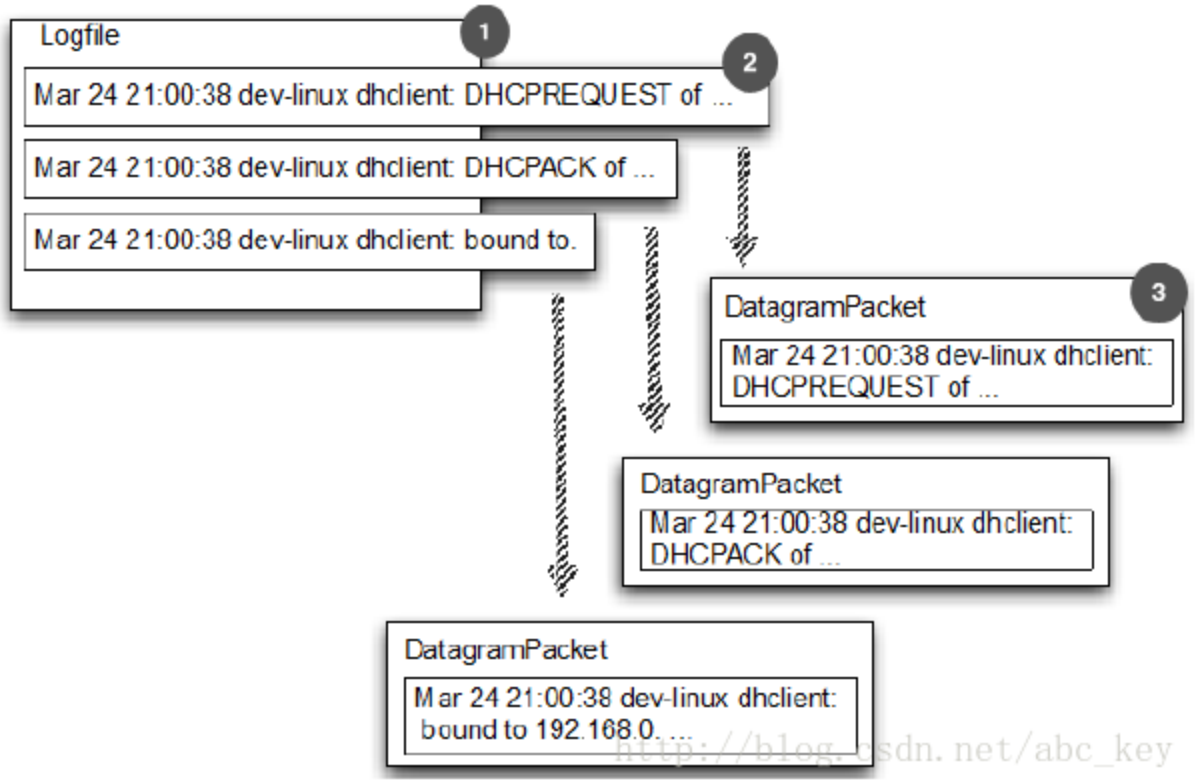
```
1.  package netty.in.action.udp;
2.
3.  import java.net.InetSocketAddress;
4.
5.  public class LogEvent {
6.
7.      public static final byte SEPARATOR = (byte) '|';
8.
9.      private final InetSocketAddress source;
10.     private final String logfile;
11.     private final String msg;
12.     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 logfil
    e, 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. }
```

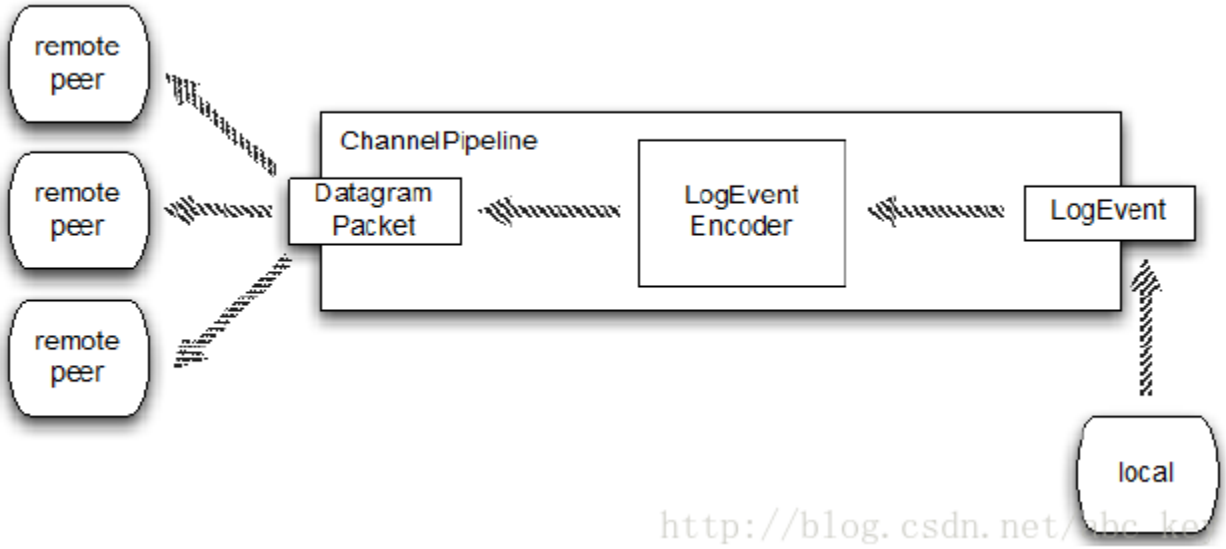
接下来的章节，我们将用这个 POJO 类来实现具体的逻辑。

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] view plaincopy

```java
package netty.in.action.udp;

import io.netty.buffer.ByteBuf;
import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.socket.DatagramPacket;
import io.netty.handler.codec.MessageToMessageEncoder;
import io.netty.util.CharsetUtil;

import java.net.InetSocketAddress;
import java.util.List;

public class LogEventEncoder extends MessageToMessageEncoder<LogEvent> {


    private final InetSocketAddress remoteAddress;

    public LogEventEncoder(InetSocketAddress remoteAddress){
        this.remoteAddress = remoteAddress;
    }

    @Override
    protected void encode(ChannelHandlerContext ctx, LogEvent msg, List<Object> out)
            throws Exception {
        ByteBuf buf = ctx.alloc().buffer();
        buf.writeBytes(msg.getLogfile().getBytes(CharsetUtil.UTF_8));
        buf.writeByte(LogEvent.SEPARATOR);
        buf.writeBytes(msg.getMsg().getBytes(CharsetUtil.UTF_8));
        out.add(new DatagramPacket(buf, remoteAddress));
    }

}
```

下面我们再编写一个广播器：

[java] view plaincopy

```java
package netty.in.action.udp;

import io.netty.bootstrap.Bootstrap;
```

```
4. import io.netty.channel.Channel;
5. import io.netty.channel.ChannelOption;
6. import io.netty.channel.EventLoopGroup;
7. import io.netty.channel.nio.NioEventLoopGroup;
8. import io.netty.channel.socket.nio.NioDatagramChannel;
9.
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();
```

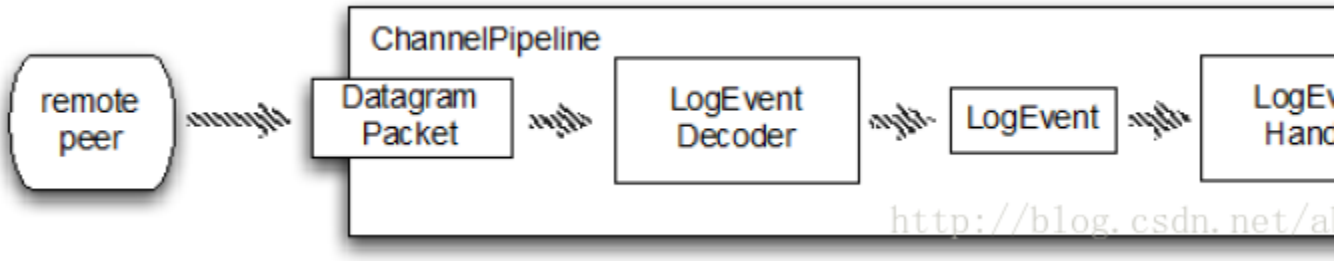
```
            }
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                Thread.interrupted();
                break;
            }
        }
    }

    public void stop() {
        group.shutdownGracefully();
    }

    public static void main(String[] args) throws Exception {
        int port = 4096;
        String path = System.getProperty("user.dir") + "/log.txt";
        LogEventBroadcaster broadcaster = new LogEventBroadcaster(new InetS
ocketAddress(
                "255.255.255.255", port), new File(path));
        try {
            broadcaster.run();
        } finally {
            broadcaster.stop();
        }
    }

}
```

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] view plaincopy

```
1. package netty.in.action.udp;
2.
3. import io.netty.buffer.ByteBuf;
4. import io.netty.channel.ChannelHandlerContext;
5. import io.netty.channel.socket.DatagramPacket;
6. import io.netty.handler.codec.MessageToMessageDecoder;
7. import io.netty.util.CharsetUtil;
8.
9. import java.util.List;
10.
11. public class LogEventDecoder extends MessageToMessageDecoder<DatagramPack
    et> {
12.
13.     @Override
14.     protected void decode(ChannelHandlerContext ctx, DatagramPacket msg, L
    ist<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(Chars
    etUtil.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] view plaincopy

```
1. package netty.in.action.udp;
2.
```

```java
import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.SimpleChannelInboundHandler;

public class LogEventHandler extends SimpleChannelInboundHandler<LogEvent> {

    @Override
    protected void channelRead0(ChannelHandlerContext ctx, LogEvent msg) throws Exception {
        StringBuilder builder = new StringBuilder();
        builder.append(msg.getReceived());
        builder.append(" [");
        builder.append(msg.getSource().toString());
        builder.append("] [");
        builder.append(msg.getLogfile());
        builder.append("] : ");
        builder.append(msg.getMsg());
        System.out.println(builder.toString());
    }
}
```

EventLogMonitor 代码如下：

[java] view plaincopy

```java
package netty.in.action.udp;

import io.netty.bootstrap.Bootstrap;
import io.netty.channel.Channel;
import io.netty.channel.ChannelInitializer;
import io.netty.channel.ChannelOption;
import io.netty.channel.ChannelPipeline;
import io.netty.channel.EventLoopGroup;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.nio.NioDatagramChannel;

import java.net.InetSocketAddress;

public class LogEventMonitor {

    private final EventLoopGroup group;
    private final Bootstrap bootstrap;

    public LogEventMonitor(InetSocketAddress address) {
        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 Ex
    ception {
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。

第十四章：实现自定义的编码解码器

本章讲述 Netty 中如何轻松实现定制的编解码器，由于 Netty 架构的灵活性，这些编解码器易于重用和测试。为了更容易实现，使用 Memcached 作为协议例子是因为它更方便我们实现。

Memcached 是免费开源、高性能、分布式的内存对象缓存系统，其目的是加速动态 Web 应用程序的响应，减轻数据库负载；Memcache 实际上是一个以 key-value 存储任意数据的内存小块。可能有人会问“为什么使用 Memcached？”，因为 Memcached 协议非常简单，便于讲解。

14.1 编解码器的范围

我们将只实现 Memcached 协议的一个子集，这足够我们进行添加、检索、删除对象；在 Memcached 中是通过执行 SET,GET,DELETE 命令来实现的。Memcached 支持很多其他的命令，但我们只使用其中三个命令，简单的东西，我们才会理解的更清楚。

Memcached 有一个二进制和纯文本协议，它们都可以用来与 Memcached 服务器通信，使用什么类型的协议取决于服务器支持哪些协议。本章主要关注实现二进制协议，因为二进制在网络编程中最常用。

14.2 实现 Memcached 的编解码器

当想要实现一个给定协议的编解码器，我们应该花一些事件来了解它的运作原理。通常情况下，协议本身都有一些详细的记录。在这里你会发现多少细节？幸运的是 Memcached 的二进制协议可以很好的扩展。

在 RFC 中有相应的规范，并提供了 Memcached 二进制协议下载地址：http://code.google.com/p/memcached/wiki/BinaryProtocolRevamped 。我们不会执行 Memcached 的所有命令，只会执行三种操作：SET,GET 和 DELETE。这样做事为了让事情变得简单。

14.3 了解 Memcached 二进制协议

可以在 http://code.google.com/p/memcached/wiki/BinaryProtocolRevamped 上详细

了解 Memcached 二进制协议结构。不过这个网站如果不翻墙的话好像访问不了。

14.4 Netty 编码器和解码器

14.4.1 实现 Memcached 编码器

先定义 memcached 操作码(Opcode)和响应状态码(Status)：

[java] view plaincopy

```java
package netty.in.action.mem;

/**
 * memcached operation codes
 * @author c.king
 *
 */
public class Opcode {

    public static final byte GET = 0x00;
    public static final byte SET = 0x01;
    public static final byte DELETE = 0x04;

}
```

[java] view plaincopy

```java
package netty.in.action.mem;

/**
 * memcached response statuses
 * @author c.king
 *
 */
public class Status {

    public static final short NO_ERROR = 0x0000;
    public static final short KEY_NOT_FOUND = 0x0001;
    public static final short KEY_EXISTS = 0x0002;
    public static final short VALUE_TOO_LARGE = 0x0003;
    public static final short INVALID_ARGUMENTS = 0x0004;
    public static final short ITEM_NOT_STORED = 0x0005;
    public static final short INC_DEC_NON_NUM_VAL = 0x0006;

}
```

继续编写 memcached 请求消息体：

[java] view plaincopy

```java
package netty.in.action.mem;

import java.util.Random;

/**
 * memcached request message object
 * @author c.king
 *
 */
public class MemcachedRequest {

    private static final Random rand = new Random();
    private int magic = 0x80;// fixed so hard coded
    private byte opCode; // the operation e.g. set or get
    private String key; // the key to delete, get or set
    private int flags = 0xdeadbeef; // random
    private int expires; // 0 = item never expires
    private String body; // if opCode is set, the value
    private int id = rand.nextInt(); // Opaque
    private long cas; // data version check...not used
    private boolean hasExtras; // not all ops have extras

    public MemcachedRequest(byte opcode, String key, String value) {
        this.opCode = opcode;
        this.key = key;
        this.body = value == null ? "" : value;
        // only set command has extras in our example
        hasExtras = opcode == Opcode.SET;
    }

    public MemcachedRequest(byte opCode, String key) {
        this(opCode, key, null);
    }

    public int getMagic() {
        return magic;
    }

    public byte getOpCode() {
        return opCode;
    }
```

```
42.
43.     public String getKey() {
44.         return key;
45.     }
46.
47.     public int getFlags() {
48.         return flags;
49.     }
50.
51.     public int getExpires() {
52.         return expires;
53.     }
54.
55.     public String getBody() {
56.         return body;
57.     }
58.
59.     public int getId() {
60.         return id;
61.     }
62.
63.     public long getCas() {
64.         return cas;
65.     }
66.
67.     public boolean isHasExtras() {
68.         return hasExtras;
69.     }
70.
71. }
```

最后编写 memcached 请求编码器：

[java] view plaincopy

```
1.  package netty.in.action.mem;
2.
3.  import io.netty.buffer.ByteBuf;
4.  import io.netty.channel.ChannelHandlerContext;
5.  import io.netty.handler.codec.MessageToByteEncoder;
6.  import io.netty.util.CharsetUtil;
7.
8.  /**
9.   * memcached request encoder
10.  * @author c.king
```

```
 *
 */
public class MemcachedRequestEncoder extends MessageToByteEncoder<MemcachedRequest> {

    @Override
    protected void encode(ChannelHandlerContext ctx, MemcachedRequest msg, ByteBuf out)
            throws Exception {
        // convert key and body to bytes array
        byte[] key = msg.getKey().getBytes(CharsetUtil.UTF_8);
        byte[] body = msg.getBody().getBytes(CharsetUtil.UTF_8);
        // total size of body = key size + body size + extras size
        int bodySize = key.length + body.length + (msg.isHasExtras() ? 8 : 0);
        // write magic int
        out.writeInt(msg.getMagic());
        // write opcode byte
        out.writeByte(msg.getOpCode());
        // write key length (2 byte) i.e a Java short
        out.writeShort(key.length);
        // write extras length (1 byte)
        int extraSize = msg.isHasExtras() ? 0x08 : 0x0;
        out.writeByte(extraSize);
        // byte is the data type, not currently implemented in Memcached
        // but required
        out.writeByte(0);
        // next two bytes are reserved, not currently implemented
        // but are required
        out.writeShort(0);
        // write total body length ( 4 bytes - 32 bit int)
        out.writeInt(bodySize);
        // write opaque ( 4 bytes) - a 32 bit int that is returned
        // in the response
        out.writeInt(msg.getId());
        // write CAS ( 8 bytes)
        // 24 byte header finishes with the CAS
        out.writeLong(msg.getCas());
        if(msg.isHasExtras()){
            // write extras
            // (flags and expiry, 4 bytes each), 8 bytes total
            out.writeInt(msg.getFlags());
            out.writeInt(msg.getExpires());
        }
```

```
52.         //write key
53.         out.writeBytes(key);
54.         //write value
55.         out.writeBytes(body);
56.     }
57.
58. }
```

14.4.2 实现 Memcached 解码器

编写 memcached 响应消息体：

[java] view plaincopy

```
1.  package netty.in.action.mem;
2.
3.  /**
4.   * memcached response message object
5.   * @author c.king
6.   *
7.   */
8.  public class MemcachedResponse {
9.
10.     private byte magic;
11.     private byte opCode;
12.     private byte dataType;
13.     private short status;
14.     private int id;
15.     private long cas;
16.     private int flags;
17.     private int expires;
18.     private String key;
19.     private String data;
20.
21.     public MemcachedResponse(byte magic, byte opCode, byte dataType, short s
    tatus,
22.             int id, long cas, int flags, int expires, String key, String dat
    a) {
23.         this.magic = magic;
24.         this.opCode = opCode;
25.         this.dataType = dataType;
26.         this.status = status;
27.         this.id = id;
28.         this.cas = cas;
29.         this.flags = flags;
```

```
30.         this.expires = expires;
31.         this.key = key;
32.         this.data = data;
33.     }
34.
35.     public byte getMagic() {
36.         return magic;
37.     }
38.
39.     public byte getOpCode() {
40.         return opCode;
41.     }
42.
43.     public byte getDataType() {
44.         return dataType;
45.     }
46.
47.     public short getStatus() {
48.         return status;
49.     }
50.
51.     public int getId() {
52.         return id;
53.     }
54.
55.     public long getCas() {
56.         return cas;
57.     }
58.
59.     public int getFlags() {
60.         return flags;
61.     }
62.
63.     public int getExpires() {
64.         return expires;
65.     }
66.
67.     public String getKey() {
68.         return key;
69.     }
70.
71.     public String getData() {
72.         return data;
73.     }
```

```
74.
75. }
```

编写 memcached 响应解码器：

[java] view plaincopy

```
1.  package netty.in.action.mem;
2.
3.  import io.netty.buffer.ByteBuf;
4.  import io.netty.channel.ChannelHandlerContext;
5.  import io.netty.handler.codec.ByteToMessageDecoder;
6.  import io.netty.util.CharsetUtil;
7.
8.  import java.util.List;
9.
10. public class MemcachedResponseDecoder extends ByteToMessageDecoder {
11.
12.     private enum State {
13.         Header, Body
14.     }
15.
16.     private State state = State.Header;
17.     private int totalBodySize;
18.     private byte magic;
19.     private byte opCode;
20.     private short keyLength;
21.     private byte extraLength;
22.     private byte dataType;
23.     private short status;
24.     private int id;
25.     private long cas;
26.
27.     @Override
28.     protected void decode(ChannelHandlerContext ctx, ByteBuf in, List<Object
    > out)
29.             throws Exception {
30.         switch (state) {
31.         case Header:
32.             // response header is 24 bytes
33.             if (in.readableBytes() < 24) {
34.                 return;
35.             }
36.             // read header
37.             magic = in.readByte();
```

```
38.             opCode = in.readByte();
39.             keyLength = in.readShort();
40.             extraLength = in.readByte();
41.             dataType = in.readByte();
42.             status = in.readShort();
43.             totalBodySize = in.readInt();
44.             id = in.readInt();
45.             cas = in.readLong();
46.             state = State.Body;
47.             break;
48.         case Body:
49.             if (in.readableBytes() < totalBodySize) {
50.                 return;
51.             }
52.             int flags = 0;
53.             int expires = 0;
54.             int actualBodySize = totalBodySize;
55.             if (extraLength > 0) {
56.                 flags = in.readInt();
57.                 actualBodySize -= 4;
58.             }
59.             if (extraLength > 4) {
60.                 expires = in.readInt();
61.                 actualBodySize -= 4;
62.             }
63.             String key = "";
64.             if (keyLength > 0) {
65.                 ByteBuf keyBytes = in.readBytes(keyLength);
66.                 key = keyBytes.toString(CharsetUtil.UTF_8);
67.                 actualBodySize -= keyLength;
68.             }
69.             ByteBuf body = in.readBytes(actualBodySize);
70.             String data = body.toString(CharsetUtil.UTF_8);
71.             out.add(new MemcachedResponse(magic, opCode, dataType, status,
72.                     id, cas, flags, expires, key, data));
73.             state = State.Header;
74.             break;
75.         default:
76.             break;
77.         }
78.     }
79.
80. }
```

14.5 测试编解码器

基于 netty 的编解码器都写完了，下面我们来写一个测试它的类：

[java] view plaincopy

```
1.  package netty.in.action.mem;
2.
3.  import io.netty.buffer.ByteBuf;
4.  import io.netty.channel.embedded.EmbeddedChannel;
5.  import io.netty.util.CharsetUtil;
6.
7.  import org.junit.Assert;
8.  import org.junit.Test;
9.
10. /**
11.  * test memcached encoder
12.  * @author c.king
13.  *
14.  */
15. public class MemcachedRequestEncoderTest {
16.
17.     @Test
18.     public void testMemcachedRequestEncoder() {
19.         MemcachedRequest request = new MemcachedRequest(Opcode.SET, "k1", "v
    1");
20.         EmbeddedChannel channel = new EmbeddedChannel(
21.                 new MemcachedRequestEncoder());
22.         Assert.assertTrue(channel.writeOutbound(request));
23.         ByteBuf encoded = (ByteBuf) channel.readOutbound();
24.         Assert.assertNotNull(encoded);
25.         Assert.assertEquals(request.getMagic(), encoded.readInt());
26.         Assert.assertEquals(request.getOpCode(), encoded.readByte());
27.         Assert.assertEquals(2, encoded.readShort());
28.         Assert.assertEquals((byte) 0x08, encoded.readByte());
29.         Assert.assertEquals((byte) 0, encoded.readByte());
30.         Assert.assertEquals(0, encoded.readShort());
31.         Assert.assertEquals(2 + 2 + 8, encoded.readInt());
32.         Assert.assertEquals(request.getId(), encoded.readInt());
33.         Assert.assertEquals(request.getCas(), encoded.readLong());
34.         Assert.assertEquals(request.getFlags(), encoded.readInt());
35.         Assert.assertEquals(request.getExpires(), encoded.readInt());
36.         byte[] data = new byte[encoded.readableBytes()];
37.         encoded.readBytes(data);
38.         Assert.assertArrayEquals((request.getKey() + request.getBody())
```

```
                .getBytes(CharsetUtil.UTF_8), data);
        Assert.assertFalse(encoded.isReadable());
        Assert.assertFalse(channel.finish());
        Assert.assertNull(channel.readInbound());
    }

}
```

[java] view plaincopy

```java
package netty.in.action.mem;

import io.netty.buffer.ByteBuf;
import io.netty.buffer.Unpooled;
import io.netty.channel.embedded.EmbeddedChannel;
import io.netty.util.CharsetUtil;

import org.junit.Assert;
import org.junit.Test;

/**
 * test memcached decoder
 *
 * @author c.king
 *
 */
public class MemcachedResponseDecoderTest {

    @Test
    public void testMemcachedResponseDecoder() {
        EmbeddedChannel channel = new EmbeddedChannel(
                new MemcachedResponseDecoder());
        byte magic = 1;
        byte opCode = Opcode.SET;
        byte dataType = 0;
        byte[] key = "Key1".getBytes(CharsetUtil.UTF_8);
        byte[] body = "Value".getBytes(CharsetUtil.UTF_8);
        int id = (int) System.currentTimeMillis();
        long cas = System.currentTimeMillis();
        ByteBuf buffer = Unpooled.buffer();
        buffer.writeByte(magic);
        buffer.writeByte(opCode);
        buffer.writeShort(key.length);
        buffer.writeByte(0);
        buffer.writeByte(dataType);
```

```
36.         buffer.writeShort(Status.KEY_EXISTS);
37.         buffer.writeInt(body.length + key.length);
38.         buffer.writeInt(id);
39.         buffer.writeLong(cas);
40.         buffer.writeBytes(key);
41.         buffer.writeBytes(body);
42.         Assert.assertTrue(channel.writeInbound(buffer));
43.         MemcachedResponse response = (MemcachedResponse) channel.readInbound
    ();
44.         assertResponse(response, magic, opCode, dataType, Status.KEY_EXISTS,
     0,
45.                 0, id, cas, key, body);
46.     }
47.
48.     private static void assertResponse(MemcachedResponse response, byte magi
    c,
49.             byte opCode, byte dataType, short status, int expires, int flags
    ,
50.             int id, long cas, byte[] key, byte[] body) {
51.         Assert.assertEquals(magic, response.getMagic());
52.         Assert.assertArrayEquals(key,
53.                 response.getKey().getBytes(CharsetUtil.UTF_8));
54.         Assert.assertEquals(opCode, response.getOpCode());
55.         Assert.assertEquals(dataType, response.getDataType());
56.         Assert.assertEquals(status, response.getStatus());
57.         Assert.assertEquals(cas, response.getCas());
58.         Assert.assertEquals(expires, response.getExpires());
59.         Assert.assertEquals(flags, response.getFlags());
60.         Assert.assertArrayEquals(body,
61.                 response.getData().getBytes(CharsetUtil.UTF_8));
62.         Assert.assertEquals(id, response.getId());
63.     }
64.
65. }
```

14.6 Summary

本章主要是使用 netty 写了个模拟 memcached 二进制协议的处理。至于 memcached 二进制协议具体是个啥玩意，可以单独了解，这里也没有详细说明。

第十五章：选择正确的线程模型

本章介绍

- 线程模型(thread-model)
- 事件循环(EventLoop)
- 并发(Concurrency)
- 任务执行(task execution)
- 任务调度(task scheduling)

线程模型定义了应用程序或框架如何执行你的代码，选择应用程序/框架的正确的线程模型是很重要的。Netty 提供了一个简单强大的线程模型来帮助我们简化代码，Netty 对所有的核心代码都进行了同步。所有 ChannelHandler，包括业务逻辑，都保证由一个线程同时执行特定的通道。这并不意味着 Netty 不能使用多线程，只是 Netty 限制每个连接都由一个线程处理，这种设计适用于非阻塞程序。我们没有必要去考虑多线程中的任何问题，也不用担心会抛 ConcurrentModificationException 或其他一些问题，如数据冗余、加锁等，这些问题在使用其他框架进行开发时是经常会发生的。

读完本章就会深刻理解 Netty 的线程模型以及 Netty 团队为什么会选择这样的线程模型，这些信息可以让我们在使用 Netty 时让程序由最好的性能。此外，Netty 提供的线程模型还可以让我们编写整洁简单的代码，以保持代码的整洁性；我们还会学习 Netty 团队的经验，过去使用其他的线程模型，现在我们将使用 Netty 提供的更容易更强大的线程模型来开发。

尽管本章讲述的是 Netty 的线程模型，但是我们仍然可以使用其他的线程模型；至于如何选择一个完美的线程模型应该根据应用程序的实际需求来判断。

本章假设如下：

- 你明白线程是什么以及如何使用，并有使用线程的工作经验；若不是这样，就请花些时间来了解清楚这些知识。推荐一本书：Java 并发编程实战。
- 你了解多线程应用程序及其设计，也包括如何保证线程安全和获取最佳性能。
- 你了解 java.util.concurrent 以及 ExecutorService 和 ScheduledExecutorService。

15.1 线程模型概述

本节将简单介绍一般的线程模型，Netty 中如何使用指定的线程模型，以及 Netty 不同的版本中使用的线程模型。你会更好的理解不同的线程模型的所有利弊。

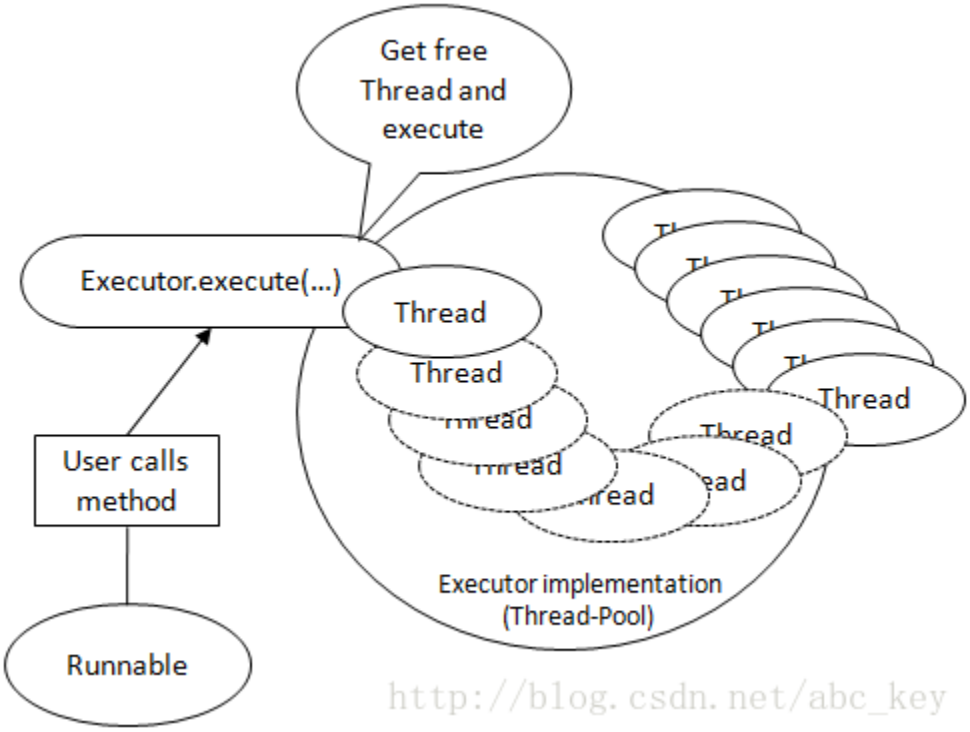
如果思考一下，在我们的生活中会发现很多情况都会使用线程模型。例如，你有一个餐厅，向你的客户提供食品，食物需要在厨房煮熟后才能给客户；某个客户下了订单后，你需要将煮熟事物这个任务发送到厨房，而厨房可以以不同的方式来处理，这就像一个线程模型，定义了如何执行任务。

- 只有一个厨师：
    - 这种方法是单线程的，一次只执行一个任务，完成当前订单后再处理下一个。
- 你有多个厨师，每个厨师都可以做，空闲的厨师准备着接单做饭：
    - 这种方式是多线程的，任务由多个线程(厨师)执行，可以并行同时执行。
- 你有多个厨师并分成组，一组做晚餐，一个做其他：

- 这种情况也是多线程，但是带有额外的限制；同时执行多个任务是由实际执行的任务类型(晚餐或其他)决定。

从上面的例子看出，日常活动适合在一个线程模型。但是 Netty 在这里适用吗？不幸的是，它没有那么简单，Netty 的核心是多线程，但隐藏了来自用户的大部分。Netty 使用多个线程来完成所有的工作，只有一个线程模型线型暴露给用户。大多数现代应用程序使用多个线程调度工作，让应用程序充分使用系统的资源来有效工作。在早期的 Java 中，这样做是通过按需创建新线程并行工作。但很快发现者不是完美的方案，因为创建和回收线程需要较大的开销。在 Java5 中加入了线程池，创建线程和重用线程交给一个任务执行，这样使创建和回收线程的开销降到最低。

下图显示使用一个线程池执行一个任务 ,提交一个任务后会使用线程池中空闲的线程来执行，完成任务后释放线程并将线程重新放回线程池：



上图每个任务线程的创建和回收不需要新线程去创建和销毁，但这只是一半的问题，我们稍后学习。你可能会问为什么不使用多线程，使用一个 ExecutorService 可以有助于防

止线程创建和回收的成本？

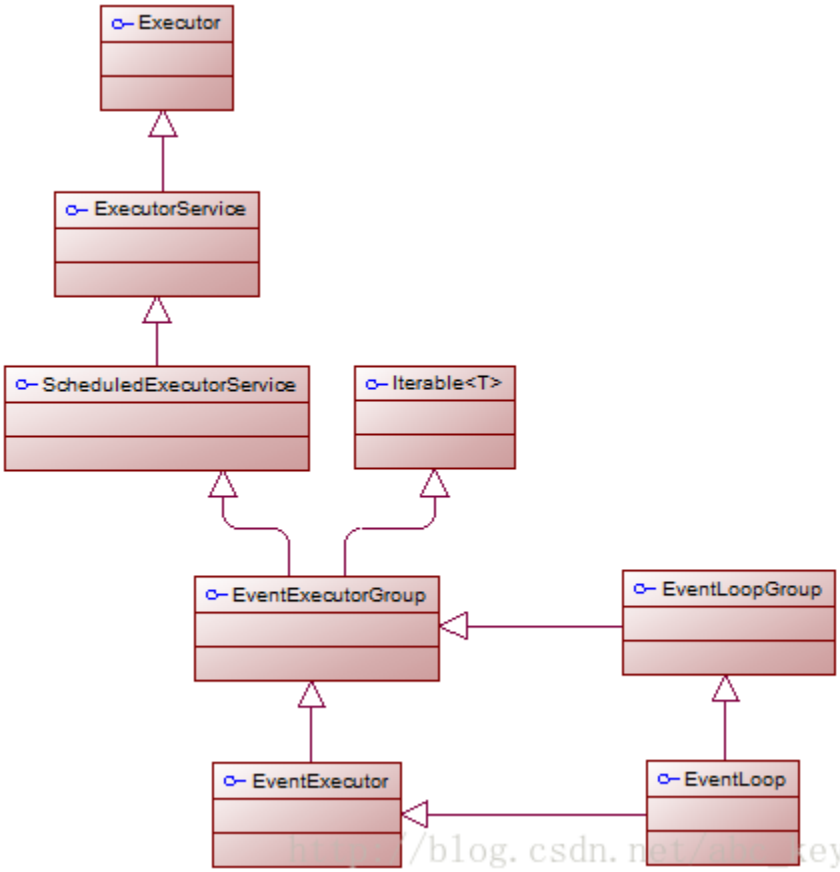
使用多线程会有太多的上下文切换，提高了资源和管理成本，这种副作用会随着运行线程的数量和执行的任务数量的增加而愈加明显。使用多线程在刚开始可能没有什么问题，但随着系统的负载增加，可能在某个点就会让系统崩溃。

除了这些技术上的限制和问题，在项目生命周期内维护应用程序/框架可能还会发生其他问题。它有效的说明了增加应用程序的复杂性取决于它是平行的，简单的陈述：编写多线程应用程序时一个辛苦的工作 !我们怎么来解决这个问题呢？在实际的场景中需要多个线程模型。让我们来看看 Netty 是如何解决这个问题的。

15.2 事件循环

事件循环所做的正如它的名字，它运行的事件在一个循环中，直到循环终止。这非常适合网络框架的设计，因为它们需要为一个特定的连接运行一个事件循环。这不是 Netty 的新发明，其他的框架和实现已经很早就这样做了。

在 Netty 中使用 EventLoop 接口代表事件循环，EventLoop 是从 EventExecutor 和 ScheduledExecutorService 扩展而来，所以可以讲任务直接交给 EventLoop 执行。类关系图如下：



15.2.1 使用事件循环

下面代码显示如何访问已分配给通道的 EventLoop 并在 EventLoop 中执行任务：

[java] view plaincopy

```
1. Channel ch = ...;
2. ch.eventLoop().execute(new Runnable() {
3.     @Override
4.     public void run() {
5.         System.out.println("run in the eventloop");
6.     }
7. });
```

使用事件循环的好处是不需要担心同步问题，在同一线程中执行所有其他关联通道的其他事件。这完全符合 Netty 的线程模型。检查任务是否已执行，使用返回的 Future，使用 Future 可以访问很多不同的操作。下面的代码是检查任务是否执行：

[java] view plaincopy

```
1. Channel ch = ...;
```

```
2.  Future<?> future = ch.eventLoop().submit(new Runnable() {
3.      @Override
4.      public void run() {
5.
6.      }
7.  });
8.  if(future.isDone()){
9.      System.out.println("task complete");
10. }else {
11.     System.out.println("task not complete");
12. }
```

检查执行任务是否在事件循环中:

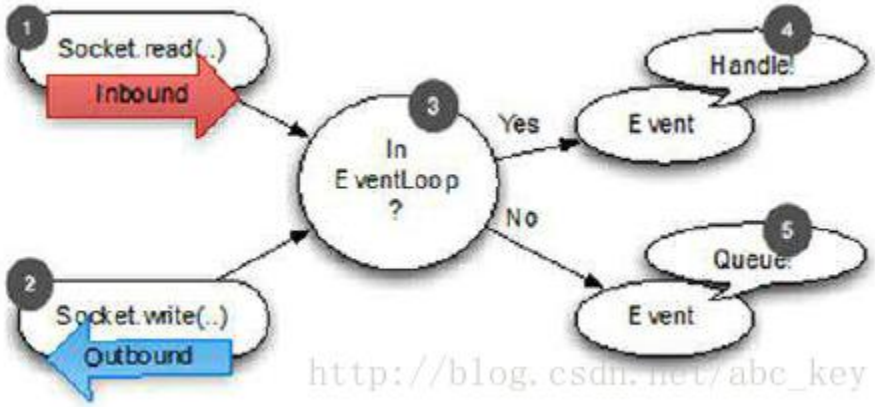
[java] view plaincopy

```
1.  Channel ch = ...;
2.  if(ch.eventLoop().inEventLoop()){
3.      System.out.println("in the EventLoop");
4.  }else {
5.      System.out.println("outside the EventLoop");
6.  }
```

只有确认没有其他 EventLoop 使用线程池了才能关闭线程池，否则可能会产生未定义的副作用。

15.2.2 Netty4 中的 I/O 操作

这个实现很强大，甚至 Netty 使用它来处理底层 I/O 事件，在 socket 上触发读和写操作。这些读和写操作是网络 API 的一部分，通过 java 和底层操作系统提供。下图显示在 EventLoop 上下文中执行入站和出站操作，如果执行线程绑定到 EventLoop，操作会直接执行；如果不是，该线程将排队执行：

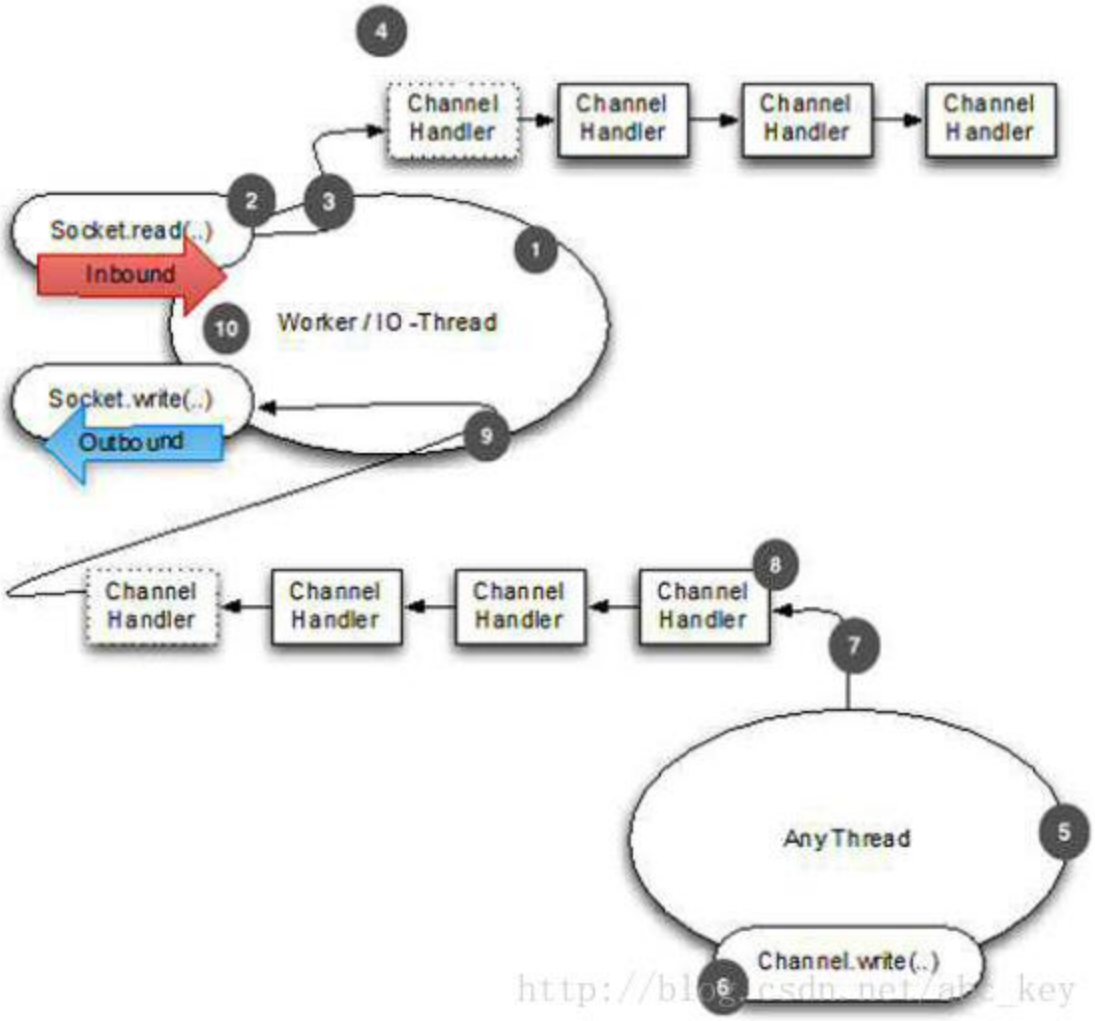


需要一次处理一个事件取决于事件的性质，通常从网络堆栈读取或传输数据到你的应用程序，有时在另外的方向做同样的事情，例如从你的应用程序传输数据到网络堆栈再发送到远程对等通道，但不限于这种类型的事物；更重要的是使用的逻辑是通用的，灵活处理各种各样的案例。

应该指出的是，线程模型(事件循环的顶部)描述并不总是由 Netty 使用。我们在了解 Netty3 后会更容易理解为什么新的线程模型是可取的。

15.2.3 Netty3 中的 I/O 操作

在以前的版本有点不同，Netty 保证在 I/O 线程中只有入站事件才被执行，所有的出站时间被调用线程处理。这看起来是个好方案，但很容易出错。它还将负责同步 ChannelHandler 来处理这些事件，因为它不保证只有一个线程同时操作；这可能发生在你去掉通道下游事件的同时，例如，在不同的线程调用 Channel.write(...)。下图显示 Netty3 的执行流程：



除了需要负担同步 ChannelHandler，这个线程模型的另一个问题是你可能需要去掉一个入站事件作为一个出站事件的结果，例如 Channel.write(...)操作导致异常。在这种情况下，捕获的异常必须生成并抛出去。乍看之下这不像是一个问题，但我们知道，捕获异常由入站事件涉及，会让你知道问题出在哪里。问题是，事实上，你现在的情况是在调用线程上执行，但捕获到异常事件必须交给工作线程来执行。这是可行的，但如果你忘了传递过去，它会导致线程模型失效；假设入站事件只有一个线程不是真，这可能会给你各种各样的竞争条件。

以前的实现有一个唯一的积极影响，在某些情况下它可以提供更好的延迟；成本是值得的，因为它消除了复杂性。实际上，在大多数应用程序中，你不会遵守任何差异延迟，还取决于其他因数，如：

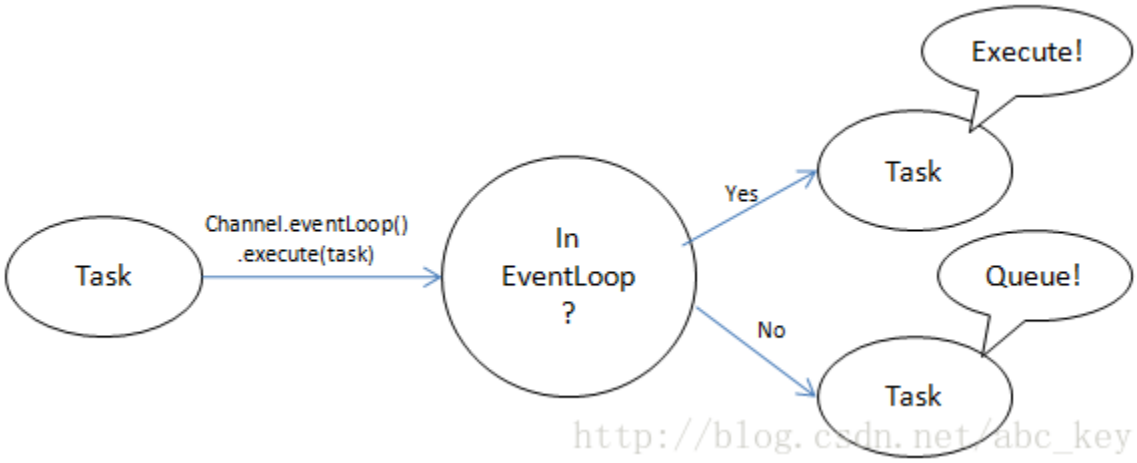
- 字节写入到远程对等通道有多快
- I/O 线程是否繁忙
- 上下文切换
- 锁定

你可以看到很多细节影响整体延迟。

15.2.4 Netty 线程模型内部

Netty 的内部实现使其线程模型表现优异，它会检查正在执行的线程是否是已分配给实际通道(和 EventLoop)，在 Channel 的生命周期内，EventLoop 负责处理所有的事件。如果线程是相同的 EventLoop 中的一个，讨论的代码块被执行；如果线程不同，它安排一个任务并在一个内部队列后执行。通常是通过 EventLoop 的 Channel 只执行一次下一个事件，这允许直接从任何线程与通道交互，同时还确保所有的 ChannelHandler 是线程安全，不需要担心并发访问问题。

下图显示在 EventLoop 中调度任务执行逻辑，这适合 Netty 的线程模型：



设计是非常重要的，以确保不要把任何长时间运行的任务放在执行队列中，因为长时间运行的任务会阻止其他在相同线程上执行的任务。这多少会影响整个系统依赖于 EventLoop 实现用于特殊传输的实现。传输之间的切换在你的代码库中可能没有任何改变，

重要的是 :切勿阻塞I/O线程。如果你必须做阻塞调用(或执行需要长时间才能完成的任务)，使用 EventExecutor。

下一节将讲解一个在应用程序中经常使用的功能，就是调度执行任务(定期执行)。Java 对这个需求提供了解决方案，但 Netty 提供了几个更好的方案。

15.3 调度任务执行

每隔一段时间需要调度任务执行，也许你想注册一个任务在客户端完成连接 5 分钟后执行，一个常见的用例是发送一个消息“你还活着？”到远程对等通道，如果远程对等通道没有反应，则可以关闭通道(连接)和释放资源。就像你和朋友打电话，沉默了一段时间后，你会说“你还在吗？”，如果朋友没有回复，就可能是断线或朋友睡着了；不管是什么问题，你都可以挂断电话，没有什么可等待的；你挂了电话后，收起电话可以做其他的事。

本节介绍使用强大的EventLoop实现任务调度，还会简单介绍Java API的任务调度，以方便和 Netty 比较加深理解。

15.3.1 使用普通的 Java API 调度任务

在 Java 中使用 JDK 提供的 ScheduledExecutorService 实现任务调度。使用 Executors 提供的静态方法创建 ScheduledExecutorService，有如下方法：

- newScheduledThreadPool(int)
- newScheduledThreadPool(int, ThreadFactory)
- newSingleThreadScheduledExecutor()
- newSingleThreadScheduledExecutor(ThreadFactory)

看下面代码：

[java] view plaincopy

```
1. ScheduledExecutorService executor = Executors.newScheduledThreadPool(10);
```

```
2.  ScheduledFuture<?> future = executor.schedule(new Runnable() {
3.      @Override
4.      public void run() {
5.          System.out.println("now it is 60 seconds later");
6.      }
7.  }, 60, TimeUnit.SECONDS);
8.  if(future.isDone()){
9.      System.out.println("scheduled completed");
10. }
11. //.....
12. executor.shutdown();
```

15.3.2 使用 EventLoop 调度任务

使用 ScheduledExecutorService 工作的很好，但是有局限性，比如在一个额外的线程中执行任务。如果需要执行很多任务，资源使用就会很严重；对于像 Netty 这样的高性能的网络框架来说，严重的资源使用是不能接受的。Netty 对这个问题提供了很好的方法。

Netty 允许使用 EventLoop 调度任务分配到通道，如下面代码：

[java] view plaincopy

```
1.  Channel ch = ...;
2.  ch.eventLoop().schedule(new Runnable() {
3.      @Override
4.      public void run() {
5.          System.out.println("now it is 60 seconds later");
6.      }
7.  }, 60, TimeUnit.SECONDS);
```

如果想任务每隔多少秒执行一次，看下面代码：

[java] view plaincopy

```
1.  Channel ch = ...;
2.  ScheduledFuture<?> future = ch.eventLoop().scheduleAtFixedRate(new Runnable(
    ) {
3.      @Override
4.      public void run() {
5.          System.out.println("after run 60 seconds,and run every 60 seconds");

6.      }
7.  }, 60, 60, TimeUnit.SECONDS);
```

```
// cancel the task
future.cancel(false);
```

15.3.3 调度的内部实现

Netty 内部实现其实是基于 George Varghese 提出的“Hashed and hierarchical timing wheels: Data structures to efficiently implement timer facility(散列和分层定时轮：数据结构有效实现定时器)”。这种实现只保证一个近似执行，也就是说任务的执行可能不是 100%准确；在实践中，这已经被证明是一个可容忍的限制，不影响多数应用程序。所以，定时执行任务不可能 100%准确的按时执行。

为了更好的理解它是如何工作，我们可以这样认为：

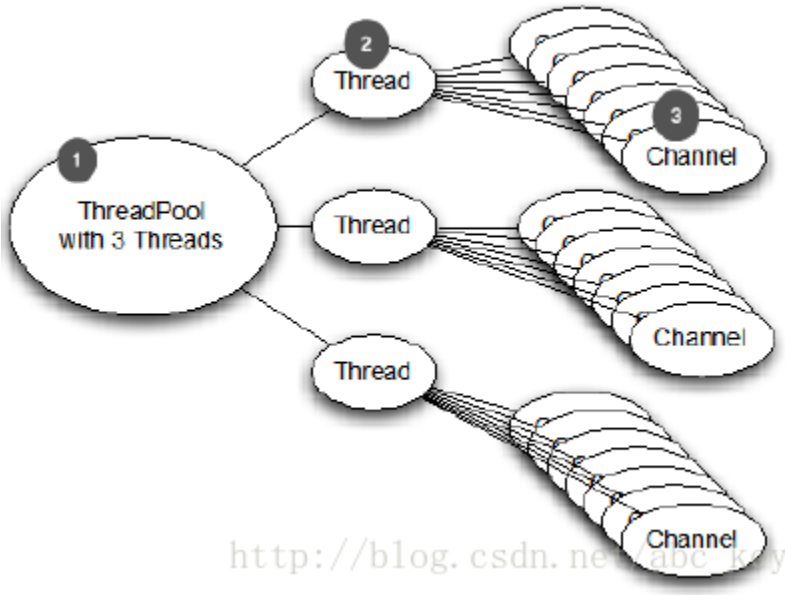
1. 在指定的延迟时间后调度任务；
2. 任务被插入到 EventLoop 的 Schedule-Task-Queue(调度任务队列)；
3. 如果任务需要马上执行，EventLoop 检查每个运行；
4. 如果有一个任务要执行，EventLoop 将立刻执行它，并从队列中删除；
5. EventLoop 等待下一次运行，从第 4 步开始一遍又一遍的重复。

因为这样的实现计划执行不可能 100%正确，对于多数用例不可能 100%准备的执行计划任务；在 Netty 中，这样的工作几乎没有资源开销。但是如果需要更准确的执行呢？很容易，你需要使用 ScheduledExecutorService 的另一个实现，这不是 Netty 的内容。记住，如果不遵循 Netty 的线程模型协议，你将需要自己同步并发访问。

15.4 I/O 线程分配细节

Netty 使用线程池来为 Channel 的 I/O 和事件服务，不同的传输实现使用不同的线程分配方式；异步实现是只有几个线程给通道之间共享，这样可以使用最小的线程数为很多的平道服务，不需要为每个通道都分配一个专门的线程。

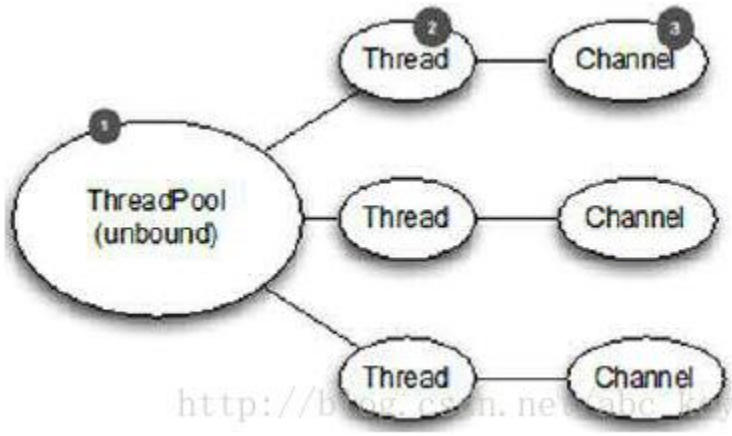
下图显示如何分配线程池：



如上图所示，使用一个固定大小的线程池管理三个线程，创建线程池后就把线程分配给线程池，确保在需要的时候，线程池中有可用的线程。这三个线程会分配给每个新创建的已连接通道，这是通过 EventLoopGroup 实现的，使用线程池来管理资源；实际会平均分配通道到所有的线程上，这种分布以循环的方式完成，因此它可能不会 100%准确，但大部分时间是准确的。

一个通道分配到一个线程后，在这个通道的生命周期内都会一直使用这个线程。这一点在以后的版本中可能会被改变，所以我们不应该依赖这种方式；不会被改变的是一个线程在同一时间只会处理一个通道的 I/O 操作，我们可以依赖这种方式，因为这种方式可以确保不需要担心同步。

下图显示 OIO(Old Blocking I/O)传输：



从上图可以看出，每个通道都有一个单独的线程。我们可以使用 java.io.*包里的类来开发基于阻塞 I/O 的应用程序，即使语义改变了，但有一件事仍然保持不变，每个通道的 I/O 在同时只能被一个线程处理；这个线程是由 Channel 的 EventLoop 提供，我们可以依靠这个硬性的规则，这也是 Netty 框架比其他网络框架更容易编写的原因。

15.5 Summary

本章主要讲解 Netty 的线程模型，其核心接口是 EventLoop；并和 OIO 中的线程模型做了比较，以突显 Netty 的优异性。

第十六章：从 EventLoop 取消注册和重新注册

本章介绍

- EventLoop
- 从 EventLoop 注册和取消注册
- 在 Netty 中使用旧的 Socket 和 Channel

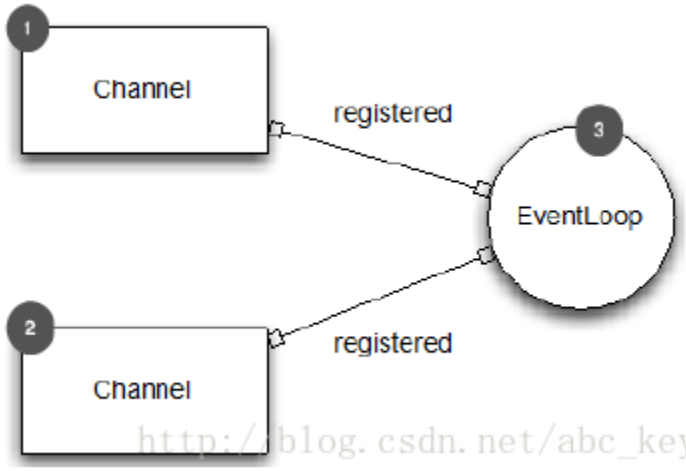
Netty 提供了一个简单的方法来连接 Socket/Channel，这是在 Netty 之外创建并转移他们的责任到 Netty。这允许你将遗留的集成框架以无缝方式一步一步迁移到 Netty；Netty 还允许取消注册的通道来停止处理 IO，这可以暂停程序处理并释放资源。

这些功能在某些情况或某种程度上可能不是非常有用，但使用这些特性可以解决一些困难的问题。举个例子，有一个非常受欢迎的社交网络，其用户增长非常快，系统程序需要处理每秒几千个交互或消息，如果用户持续增长，系统将会处理每秒数以万计的交互；这很令人兴奋，但随着用户数的增长，系统将消耗大量的内存和 CPU 而导致性能低下；此时最需要做的就是改进他们，并且不要花太多的钱在硬件设备上。这种情况下，系统必须保持功能正常能处理日益增长的数据量，此时，注册/注销事件循环就派上用场了。

通过允许外部 Socket/Channel 来注册和注销，Netty 能够以这样的方式改进旧系统的缺陷，所有的 Netty 程序都可以通过一种有效精巧的方式整合到现有系统，本章将重点讲解 Netty 是如何整合。

16.1 注册和取消注册的 Channel 和 Socket

前面章节讲过，每个通道需要注册到一个 EventLoop 来处理 IO 或事件，这是在引导过程中自动完成。下图显示了他们的关系：



上图只是显示了他们关系的一部分，通道关闭时，还需要将注册到 EventLoop 中的 Socket/Channel 注销以释放资源。

有时不得不处理 java.nio.channels.SocketChannel 或其他 java.nio.channes.Channel 实现，这可能是遗留程序或框架的一些原因所致。我们可以使用 Netty 来包装预先创建的 java.nio.channels.Channel，然后再注册到 EventLoop。我们可以使用 Netty 的所有特性，同时还能重用现有的东西。下面代码显示了此功能：

[java] view plaincopy

```java
//nio
java.nio.channels.SocketChannel mySocket = java.nio.channels.SocketChannel.open();
//netty
SocketChannel ch = new NioSocketChannel(mySocket);
```

```
5.  EventLoopGroup group = new NioEventLoopGroup();
6.  //register channel
7.  ChannelFuture registerFuture = group.register(ch);
8.  //de-register channel
9.  ChannelFuture deregisterFuture = ch.deregister();
```

Netty 也适用于包装 OIO，看下面代码：

[java] view plaincopy

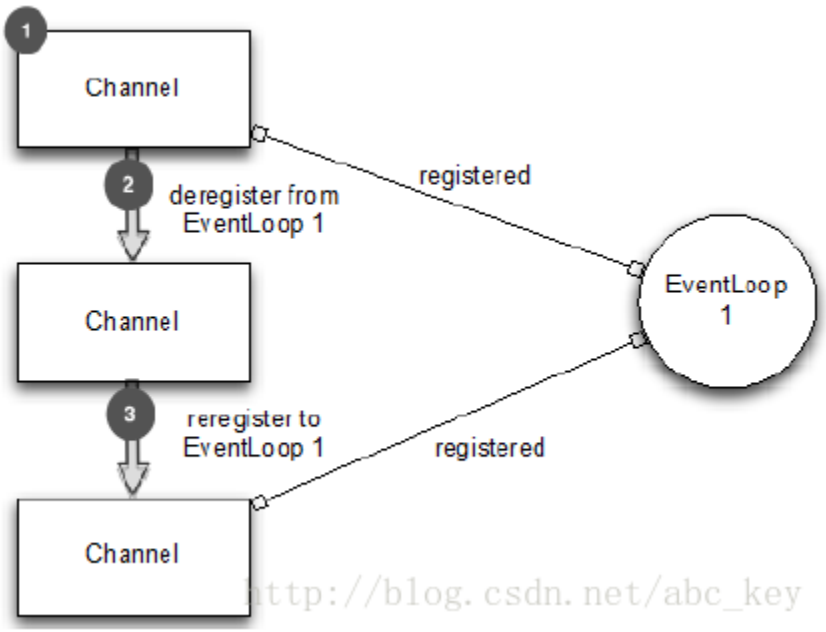
```
1.  //oio
2.  Socket mySocket = new Socket("www.baidu.com", 80);
3.  //netty
4.  SocketChannel ch = new OioSocketChannel(mySocket);
5.  EventLoopGroup group = new OioEventLoopGroup();
6.  //register channel
7.  ChannelFuture registerFuture = group.register(ch);
8.  //de-register channel
9.  ChannelFuture deregisterFuture = ch.deregister();
```

只有 2 个重点如下：

1. 使用 Netty 包装已创建的 Socket 或 Channel 必须使用与之对应的实现，如 Socket 是 OIO，则使用 Netty 的 OioSocketChannel；SocketChannel 是 NIO，则使用 NioSocketChannel。
2. EventLoop.register(...)和 Channel.deregister(...)都是非阻塞异步的，也就是说它们可能不会理解执行完成，可能稍后完成。它们返回 ChannelFuture，我们在需要进一步操作或确认完成操作时可以添加一个 ChannelFutureLister 或在 ChannelFuture 上同步等待至完成；选择哪一种方式看实际需求，一般建议使用 ChannelFutureLister，应避免阻塞。

16.2 挂起 IO 处理

在一些情况下可能需要停止一个指定通道的处理操作，比如程序耗尽内存、崩溃、失去一些消息，此时，我们可以停止处理事件的通道来清理系统资源，以保持程序稳定继续处理后续消息。若这样做，最好的方式就是从 EventLoop 取消注册的通道，这可以有效阻止通道再处理任何事件。若需要被取消的通道再次处理事件，则只需要将该通道重新注册到 EventLooop 即可。看下图：



看下面代码：

[java] view plaincopy

```java
EventLoopGroup group = new NioEventLoopGroup();
Bootstrap bootstrap = new Bootstrap();
bootstrap.group(group).channel(NioSocketChannel.class)
        .handler(new SimpleChannelInboundHandler<ByteBuf>() {
            @Override
            protected void channelRead0(ChannelHandlerContext ctx,
                    ByteBuf msg) throws Exception {
                //remove this ChannelHandler and de-register
                ctx.pipeline().remove(this);
                ctx.deregister();
            }
        });
ChannelFuture future = bootstrap.connect(
        new InetSocketAddress("www.baidu.com", 80)).sync();
//....
Channel channel = future.channel();
//re-register channel and add ChannelFutureLister
group.register(channel).addListener(new ChannelFutureListener() {
    @Override
    public void operationComplete(ChannelFuture future) throws Exception {
        if(future.isSuccess()){
            System.out.println("Channel registered");
        }else{
            System.out.println("register channel on EventLoop fail");
            future.cause().printStackTrace();
```

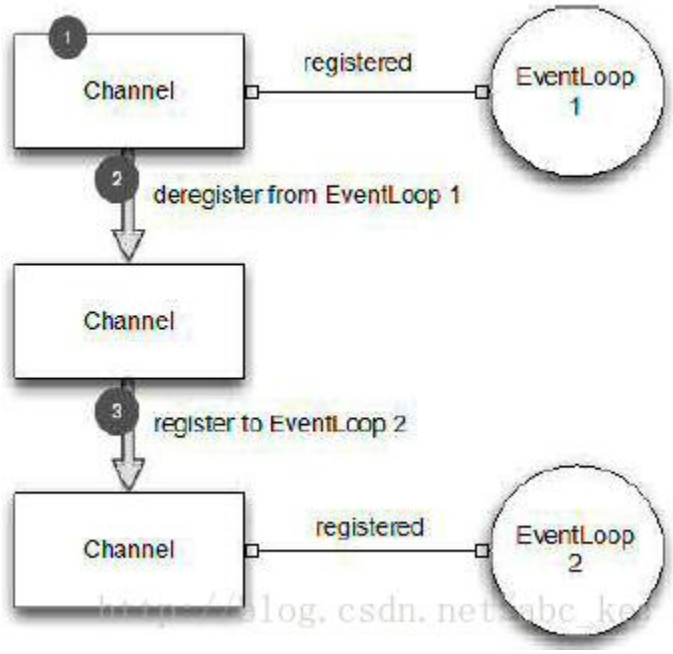
```
        }
    }
});
```

16.3 迁移通道到另一个事件循环

另一个取消注册和注册一个 Channel 的用例是将一个活跃的 Channel 移到另一个 EventLoop，有下面一些原因可能导致需要这么做：

- 当前 EventLoop 太忙碌，需要将 Channel 移到一个不是很忙碌的 EventLoop；
- 终止 EventLoop 释放资源同时保持活跃 Channel 可以继续使用；
- 迁移 Channel 到一个执行级别较低的非关键业务的 EventLoop 中。

下图显示迁移 Channel 到另一个 EventLoop：



看下面代码：

[java] view plaincopy

```
EventLoopGroup group = new NioEventLoopGroup();
final EventLoopGroup group2 = new NioEventLoopGroup();
Bootstrap b = new Bootstrap();
b.group(group).channel(NioSocketChannel.class)
        .handler(new SimpleChannelInboundHandler<ByteBuf>() {
            @Override
            protected void channelRead0(ChannelHandlerContext ctx,
                    ByteBuf msg) throws Exception {
                // remove this channel handler and de-register
                ctx.pipeline().remove(this);
```

```
                ChannelFuture f = ctx.deregister();
                // add ChannelFutureListener
                f.addListener(new ChannelFutureListener() {
                    @Override
                    public void operationComplete(ChannelFuture future)
                            throws Exception {
                        // migrate this handler register to group2
                        group2.register(future.channel());
                    }
                });
            }
        });
ChannelFuture future = b.connect("www.baidu.com", 80);
future.addListener(new ChannelFutureListener() {
    @Override
    public void operationComplete(ChannelFuture future)
            throws Exception {
        if (future.isSuccess()) {
            System.out.println("connection established");
        } else {
            System.out.println("connection attempt failed");
            future.cause().printStackTrace();
        }
    }
});
```

16.4 Summary

至此，netty in action 中文版系列博文已完成了，一次不经意的 baidu，发现在 51cto 上都出现本系列博客的 pdf 文件了，下载下来一看发现和本系列内容一模一样，呵呵 ，看来 netty 中文资料的需求还是有一些的。