**第1章　Netty——异步和事件驱动**

**本章主要内容**

* Java网络编程
* Netty简介
* Netty的核心组件

假设你正在为一个重要的大型公司开发一款全新的任务关键型的应用程序。在第一次会议上，你得知该系统必须要能够扩展到支撑150 000名并发用户，并且不能有任何的性能损失，这时所有的目光都投向了你。你会怎么说呢？

如果你可以自信地说：“当然，没问题。”那么大家都会向你脱帽致敬。但是，我们大多数人可能会采取一个更加谨慎的立场，例如：“听上去是可行的。”然后，一回到计算机旁，我们便开始搜索“high performance Java networking”（高性能Java网络编程）。

如果你现在搜索它，在第一页结果中，你将会看到下面的内容：

Netty: Home

netty.io/

Netty是一款异步的事件驱动的**网络**应用程序框架，支持快速地开发可维护的**高性能**的面向协议的服务器和客户端。

如果你和大多数人一样，通过这样的方式发现了Netty，那么你的下一步多半是：浏览该网站，下载源代码，仔细阅读Javadoc和一些相关的博客，然后写点儿代码试试。如果你已经有了扎实的网络编程经验，那么可能进展还不错，不然则可能是一头雾水。

这是为什么呢？因为像我们例子中那样的高性能系统不仅要求超一流的编程技巧，还需要几个复杂领域（网络编程、多线程处理和并发）的专业知识。Netty优雅地处理了这些领域的知识，使得即使是网络编程新手也能使用。但到目前为止，由于还缺乏一本全面的指南，使得对它的学习过程比实际需要的艰涩得多——因此便有了这本书。

我们编写这本书的主要目的是：使得Netty能够尽可能多地被更加广泛的开发者采用。这也包括那些拥有创新的内容或者服务，却没有时间或者兴趣成为网络编程专家的人。如果这适用于你，我们相信你将会非常惊讶自己这么快便可以开始创建你的第一款基于Netty的应用程序了。当然在另一个层面上讲，我们也需要支持那些正在寻找工具来创建他们自己的网络协议的高级从业人员。

Netty确实提供了极为丰富的网络编程工具集，我们将花大部分的时间来探究它的能力。但是，Netty终究是一个框架，它的架构方法和设计原则是：每个小点都和它的技术性内容一样重要，穷其精妙。因此，我们也将探讨很多其他方面的内容，例如：

* 关注点分离——业务和网络逻辑解耦；
* 模块化和可复用性；
* 可测试性作为首要的要求。

在这第1章中，我们将从一些与高性能网络编程相关的背景知识开始铺陈，特别是它在Java开发工具包（JDK）中的实现。有了这些背景知识后，我们将介绍Netty，它的核心概念以及构建块。在本章结束之后，你就能够编写你的第一款基于Netty的客户端和服务器应用程序了。

**1.1　Java网络编程**

早期的网络编程开发人员，需要花费大量的时间去学习复杂的C语言套接字库，去处理它们在不同的操作系统上出现的古怪问题。虽然最早的Java（1995—2002）引入了足够多的面向对象façade（门面）来隐藏一些棘手的细节问题，但是创建一个复杂的客户端/服务器协议仍然需要大量的样板代码（以及相当多的底层研究才能使它整个流畅地运行起来）。

那些最早期的Java API（java.net）只支持由本地系统套接字库提供的所谓的阻塞函数。代码清单1-1展示了一个使用了这些函数调用的服务器代码的普通示例。

代码清单1-1 阻塞I/O示例

ServerSocket serverSocket = new ServerSocket(portNumber); ⇽ -- 创建一个新的ServerSocket，用以监听指定端口上的连接请求

Socket clientSocket = serverSocket.accept(); ⇽ -- ❶ 对accept()方法的调用将被阻塞，直到一个连接建立

BufferedReader in = new BufferedReader(

new InputStreamReader(clientSocket.getInputStream()));

PrintWriter out =

new PrintWriter(clientSocket.getOutputStream(), true); ⇽ -- ❷ 这些流对象都派生于该套接字的流对象

String request, response;

while ((request = in.readLine()) != null) { ⇽ -- ❸ 处理循环开始

if ("Done".equals(request)) {

break; ⇽ -- 如果客户端发送了“Done”，则退出处理循环

}

response = processRequest(request); ⇽ -- ❹ 请求被传递给服

务器的处理方法

out.println(response); ⇽ -- 服务器的响应被发送给了客户端

} ⇽ -- 继续执行处理循环

44代码清单1-1实现了Socket API的基本模式之一。以下是最重要的几点。

* ServerSocket上的accept()方法将会一直阻塞到一个连接建立❶，随后返回一个新的Socket用于客户端和服务器之间的通信。该ServerSocket将继续监听传入的连接。
* BufferedReader和PrintWriter都衍生自Socket的输入输出流❷。前者从一个字符输入流中读取文本，后者打印对象的格式化的表示到文本输出流。
* readLine()方法将会阻塞，直到在❸处一个由换行符或者回车符结尾的字符串被读取。
* 客户端的请求已经被处理❹。

这段代码片段将只能同时处理一个连接，要管理多个并发客户端，需要为每个新的客户端Socket创建一个新的Thread，如图1-1所示。

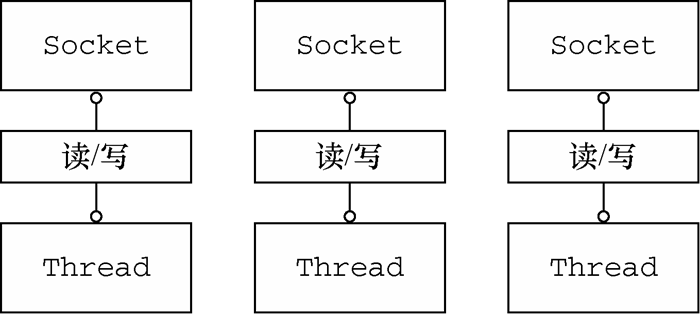


图1-1　使用阻塞I/O处理多个连接

让我们考虑一下这种方案的影响。第一，在任何时候都可能有大量的线程处于休眠状态，只是等待输入或者输出数据就绪，这可能算是一种资源浪费。第二，需要为每个线程的调用栈都分配内存，其默认值大小区间为64 KB到1 MB，具体取决于操作系统。第三，即使Java虚拟机（JVM）在物理上可以支持非常大数量的线程，但是远在到达该极限之前，上下文切换所带来的开销就会带来麻烦，例如，在达到10 000个连接的时候。

虽然这种并发方案对于支撑中小数量的客户端来说还算可以接受，但是为了支撑100 000或者更多的并发连接所需要的资源使得它很不理想。幸运的是，还有一种方案。

**1.1.1　Java NIO**

除了代码清单1-1中代码底层的阻塞系统调用之外，本地套接字库很早就提供了非阻塞调用，其为网络资源的利用率提供了相当多的控制：

* 可以使用setsockopt()方法配置套接字，以便读/写调用在没有数据的时候立即返回，也就是说，如果是一个阻塞调用应该已经被阻塞了[[1]](http://www.epubit.com.cn/book/onlinechapter/52871#anchor11)；
* 可以使用操作系统的事件通知API[[2]](http://www.epubit.com.cn/book/onlinechapter/52871#anchor12)注册一组非阻塞套接字，以确定它们中是否有任何的套接字已经有数据可供读写。

Java对于非阻塞I/O的支持是在2002年引入的，位于JDK 1.4的java.nio包中。

新的还是非阻塞的

NIO最开始是新的输入/输出（New Input/Output）的英文缩写，但是，该Java API已经出现足够长的时间了，不再是“新的”了，因此，如今大多数的用户认为NIO代表非阻塞I/O（Non-blocking I/O），而阻塞I/O（blocking I/O）是旧的输入/输出（old input/output，OIO）。你也可能遇到它被称为普通I/O（plain I/O）的时候。

**1.1.2　选择器**

图1-2展示了一个非阻塞设计，其实际上消除了上一节中所描述的那些弊端。

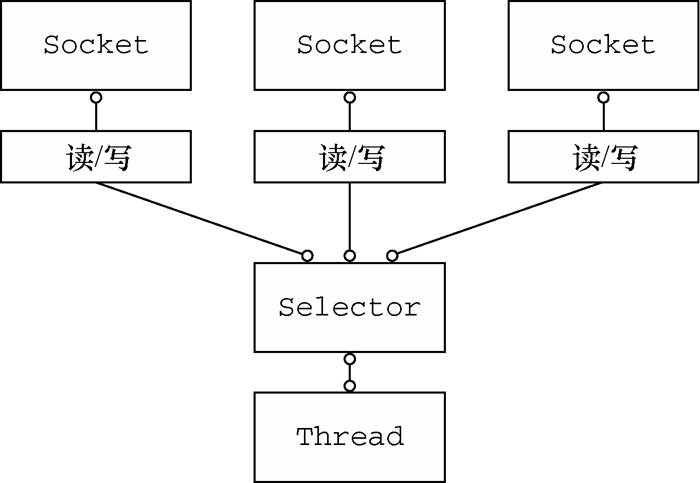


图1-2　使用Selector的非阻塞I/O

class java.nio.channels.Selector是Java的非阻塞I/O实现的关键。它使用了事件通知API以确定在一组非阻塞套接字中有哪些已经就绪能够进行I/O相关的操作。因为可以在任何的时间检查任意的读操作或者写操作的完成状态，所以如图1-2所示，一个单一**的**线程便可以处理多个并发的连接。

总体来看，与阻塞I/O模型相比，这种模型提供了更好的资源管理：

* 使用较少的线程便可以处理许多连接，因此也减少了内存管理和上下文切换所带来开销；
* 当没有I/O操作需要处理的时候，线程也可以被用于其他任务。

尽管已经有许多直接使用Java NIO API的应用程序被构建了，但是要做到如此正确和安全并不容易。特别是，在高负载下可靠和高效地处理和调度I/O操作是一项繁琐而且容易出错的任务，最好留给高性能的网络编程专家——Netty。

**1.2　Netty简介**

不久以前，我们在本章一开始所呈现的场景——支持成千上万的并发客户端——还被认定为是不可能的。然而今天，作为系统用户，我们将这种能力视为理所当然；同时作为开发人员，我们期望将水平线提得更高[[3]](http://www.epubit.com.cn/book/onlinechapter/52871#anchor13)。因为我们知道，总会有更高的吞吐量和可扩展性的要求——在更低的成本的基础上进行交付。

不要低估了这最后一点的重要性。我们已经从漫长的痛苦经历中学到：直接使用底层的API暴露了复杂性，并且引入了对往往供不应求的技能的关键性依赖[[4]](http://www.epubit.com.cn/book/onlinechapter/52871#anchor14)。这也就是，面向对象的基本概念：用较简单的抽象隐藏底层实现的复杂性。

这一原则也催生了大量框架的开发，它们为常见的编程任务封装了解决方案，其中的许多都和分布式系统的开发密切相关。我们可以确定地说：所有专业的Java开发人员都至少对它们熟知一二。[[5]](http://www.epubit.com.cn/book/onlinechapter/52871#anchor15)对于我们许多人来说，它们已经变得不可或缺，因为它们既能满足我们的技术需求，又能满足我们的时间表。

在网络编程领域，Netty是Java的卓越框架。[[6]](http://www.epubit.com.cn/book/onlinechapter/52871#anchor16)它驾驭了Java高级API的能力，并将其隐藏在一个易于使用的API之后。Netty使你可以专注于自己真正感兴趣的——你的应用程序的独一无二的价值。

在我们开始首次深入地了解Netty之前，请仔细审视表1-1中所总结的关键特性。有些是技术性的，而其他的更多的则是关于架构或设计哲学的。在本书的学习过程中，我们将不止一次地重新审视它们。

表1-1　Netty的特性总结

| 分　　类 | Netty的特性 |
| --- | --- |
| 设计 | 统一的API，支持多种传输类型，阻塞的和非阻塞的简单而强大的线程模型真正的无连接数据报套接字支持链接逻辑组件以支持复用 |
| 易于使用 | 详实的Javadoc和大量的示例集不需要超过JDK 1.6+[[7]](http://www.epubit.com.cn/book/onlinechapter/52871#anchor17)的依赖。（一些可选的特性可能需要Java 1.7+和/或额外的依赖） |
| 性能 | 拥有比Java的核心API更高的吞吐量以及更低的延迟得益于池化和复用，拥有更低的资源消耗最少的内存复制 |
| 健壮性 | 不会因为慢速、快速或者超载的连接而导致OutOfMemoryError消除在高速网络中NIO应用程序常见的不公平读/写比率 |
| 安全性 | 完整的SSL/TLS以及StartTLS支持可用于受限环境下，如Applet和OSGI |
| 社区驱动 | 发布快速而且频繁 |

**1.2.1　谁在使用Netty**

Netty拥有一个充满活力并且不断壮大的用户社区，其中不乏大型公司，如Apple、Twitter、Facebook、Google、Square和Instagram，还有流行的开源项目，如Infinispan、HornetQ、Vert.x、Apache Cassandra和Elasticsearch[[8]](http://www.epubit.com.cn/book/onlinechapter/52871#anchor18)，它们所有的核心代码都利用了Netty强大的网络抽象[[9]](http://www.epubit.com.cn/book/onlinechapter/52871#anchor19)。在初创企业中，Firebase和Urban Airship也在使用Netty，前者用来做HTTP长连接，而后者用来支持各种各样的推送通知。

每当你使用Twitter，你便是在使用Finagle[[10]](http://www.epubit.com.cn/book/onlinechapter/52871#anchor110)，它们基于Netty的系统间通信框架。Facebook在Nifty中使用了Netty，它们的Apache Thrift服务。可伸缩性和性能对这两家公司来说至关重要，他们也经常为Netty贡献代码[[11]](http://www.epubit.com.cn/book/onlinechapter/52871#anchor111)。

反过来，Netty也已从这些项目中受益，通过实现FTP、SMTP、HTTP和WebSocket以及其他的基于二进制和基于文本的协议，Netty扩展了它的应用范围及灵活性。

**1.2.2　异步和事件驱动**

因为我们要大量地使用“异步”这个词，所以现在是一个澄清上下文的好时机。异步（也就是非同步）事件肯定大家都熟悉。考虑一下电子邮件：你可能会也可能不会收到你已经发出去的电子邮件对应的回复，或者你也可能会在正在发送一封电子邮件的时候收到一个意外的消息。异步事件也可以具有某种有序的关系。通常，你只有在已经问了一个问题之后才会得到一个和它对应的答案，而在你等待它的同时你也可以做点别的事情。

在日常的生活中，异步自然而然地就发生了，所以你可能没有对它考虑过多少。但是让一个计算机程序以相同的方式工作就会产生一些非常特殊的问题。本质上，一个既是异步的又是事件驱动的系统会表现出一种特殊的、对我们来说极具价值的行为：它可以以任意的顺序响应在任意的时间点产生的事件。

这种能力对于实现最高级别的可伸缩性至关重要，定义为：“一种系统、网络或者进程在需要处理的工作不断增长时，可以通过某种可行的方式或者扩大它的处理能力来适应这种增长的能力。”[[12]](http://www.epubit.com.cn/book/onlinechapter/52871#anchor112)

异步和可伸缩性之间的联系又是什么呢？

* 非阻塞网络调用使得我们可以不必等待一个操作的完成。完全异步的I/O正是基于这个特性构建的，并且更进一步：异步方法会立即返回，并且在它完成时，会直接或者在稍后的某个时间点通知用户。
* 选择器使得我们能够通过较少的线程便可监视许多连接上的事件。

将这些元素结合在一起，与使用阻塞I/O来处理大量事件相比，使用非阻塞I/O来处理更快速、更经济。从网络编程的角度来看，这是构建我们理想系统的关键，而且你会看到，这也是Netty的设计底蕴的关键。

在1.3节中，我们将首先看一看Netty的核心组件。现在，只需要将它们看作是域对象，而不是具体的Java类。随着时间的推移，我们将看到它们是如何协作，来为在网络上发生的事件提供通知，并使得它们可以被处理的。

**1.3　Netty的核心组件**

在本节中我将要讨论Netty的主要构件块：

* Channel；
* 回调；
* Future；
* 事件和ChannelHandler。

这些构建块代表了不同类型的构造：资源、逻辑以及通知。你的应用程序将使用它们来访问网络以及流经网络的数据。

对于每个组件来说，我们都将提供一个基本的定义，并且在适当的情况下，还会提供一个简单的示例代码来说明它的用法。

**1.3.1　Channel**

Channel是Java NIO的一个基本构造。

它代表一个到实体（如一个硬件设备、一个文件、一个网络套接字或者一个能够执行一个或者多个不同的I/O操作的程序组件）的开放连接，如读操作和写操作[[13]](http://www.epubit.com.cn/book/onlinechapter/52871#anchor113)。

目前，可以把Channel看作是传入（入站）或者传出（出站）数据的载体。因此，它可以被打开或者被关闭，连接或者断开连接。

**1.3.2　回调**

一个回调其实就是一个方法，一个指向已经被提供给另外一个方法的方法的引用。这使得后者[[14]](http://www.epubit.com.cn/book/onlinechapter/52871#anchor114)可以在适当的时候调用前者。回调在广泛的编程场景中都有应用，而且也是在操作完成后通知相关方最常见的方式之一。

Netty在内部使用了回调来处理事件；当一个回调被触发时，相关的事件可以被一个interface-ChannelHandler的实现处理。代码清单1-2展示了一个例子：当一个新的连接已经被建立时，ChannelHandler的channelActive()回调方法将会被调用，并将打印出一条信息。

代码清单1-2　被回调触发的**ChannelHandler**

public class ConnectHandler extends ChannelInboundHandlerAdapter {

@Override

public void channelActive(ChannelHandlerContext ctx)

throws Exception { ⇽ -- 当一个新的连接已经被建立时，channelActive(ChannelHandlerContext)将会被调用

System.out.println(

"Client " + ctx.channel().remoteAddress() + " connected");

}

}

### 1.3.3　Future

Future提供了另一种在操作完成时通知应用程序的方式。这个对象可以看作是一个异步操作的结果的占位符；它将在未来的某个时刻完成，并提供对其结果的访问。

JDK预置了interface java.util.concurrent.Future，但是其所提供的实现，只允许手动检查对应的操作是否已经完成，或者一直阻塞直到它完成。这是非常繁琐的，所以Netty提供了它自己的实现——ChannelFuture，用于在执行异步操作的时候使用。

ChannelFuture提供了几种额外的方法，这些方法使得我们能够注册一个或者多个ChannelFutureListener实例。监听器的回调方法operationComplete()，将会在对应的操作完成时被调用[[15]](http://www.epubit.com.cn/book/onlinechapter/52871#anchor115)。然后监听器可以判断该操作是成功地完成了还是出错了。如果是后者，我们可以检索产生的Throwable。简而言之，由ChannelFutureListener提供的通知机制消除了手动检查对应的操作是否完成的必要。

每个Netty的出站I/O操作都将返回一个ChannelFuture；也就是说，它们都不会阻塞。正如我们前面所提到过的一样，Netty完全是异步和事件驱动的。

代码清单1-3展示了一个ChannelFuture作为一个I/O操作的一部分返回的例子。这里，connect()方法将会直接返回，而不会阻塞，该调用将会在后台完成。这究竟什么时候会发生则取决于若干的因素，但这个关注点已经从代码中抽象出来了。因为线程不用阻塞以等待对应的操作完成，所以它可以同时做其他的工作，从而更加有效地利用资源。

代码清单1-3　异步地建立连接

Channel channel = ...;

![](/api/storage/getbykey/screenshow?key=1704e6378455b23f17f7)![](/api/storage/getbykey/screenshow?key=1704725cc4b464f4793a)![](/api/storage/getbykey/screenshow?key=17042eacf3010fc856d6)// Does not block

ChannelFuture future = channel.connect( ⇽ -- 异步地连接到远程节点

new InetSocketAddress("192.168.0.1", 25));

代码清单1-4显示了如何利用ChannelFutureListener。首先，要连接到远程节点上。然后，要注册一个新的ChannelFutureListener到对connect()方法的调用所返回的ChannelFuture上。当该监听器被通知连接已经建立的时候，要检查对应的状态❶。如果该操作是成功的，那么将数据写到该Channel。否则，要从ChannelFuture中检索对应的Throwable。

代码清单1-4　回调实战

Channel channel = ...;

// Does not block

ChannelFuture future = channel.connect( ⇽ -- 异步地连接到远程节点

new InetSocketAddress("192.168.0.1", 25));

future.addListener(new ChannelFutureListener() { ⇽ -- 注册一个ChannelFutureListener，以便在操作完成时获得通知

@Override

public void operationComplete(ChannelFuture future) { ⇽ -- ❶ 检查操作

的状态

if (future.isSuccess()){

ByteBuf buffer = Unpooled.copiedBuffer( ⇽ -- 如果操作是成功的，则创建一个ByteBuf以持有数据

"Hello",Charset.defaultCharset());

ChannelFuture wf = future.channel()

.writeAndFlush(buffer); ⇽ -- 将数据异步地发送到远程节点。

返回一个ChannelFuture

....

} else {

Throwable cause = future.cause();　　⇽ --　如果发生错误，则访问描述原因的Throwable

cause.printStackTrace();

}

}

});

需要注意的是，对错误的处理完全取决于你、目标，当然也包括目前任何对于特定类型的错误加以的限制。例如，如果连接失败，你可以尝试重新连接或者建立一个到另一个远程节点的连接。

如果你把ChannelFutureListener看作是回调的一个更加精细的版本，那么你是对的。事实上，回调和Future是相互补充的机制；它们相互结合，构成了Netty本身的关键构件块之一。

**1.3.4　事件和ChannelHandler**

Netty使用不同的事件来通知我们状态的改变或者是操作的状态。这使得我们能够基于已经发生的事件来触发适当的动作。这些动作可能是：

* 记录日志；
* 数据转换；
* 流控制；
* 应用程序逻辑。

Netty是一个网络编程框架，所以事件是按照它们与入站或出站数据流的相关性进行分类的。可能由入站数据或者相关的状态更改而触发的事件包括：

* 连接已被激活或者连接失活；
* 数据读取；
* 用户事件；
* 错误事件。

出站事件是未来将会触发的某个动作的操作结果，这些动作包括：

* 打开或者关闭到远程节点的连接；
* 将数据写到或者冲刷到套接字。

每个事件都可以被分发给ChannelHandler类中的某个用户实现的方法。这是一个很好的将事件驱动范式直接转换为应用程序构件块的例子。图1-3展示了一个事件是如何被一个这样的ChannelHandler链处理的。

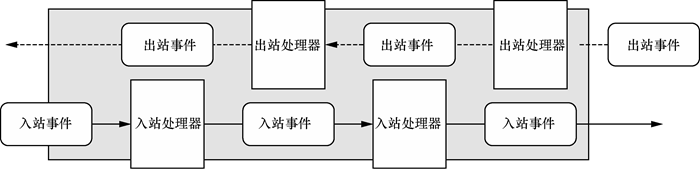


图1-3　流经ChannelHandler链的入站事件和出站事件

Netty的ChannelHandler为处理器提供了基本的抽象，如图1-3所示的那些。我们会在适当的时候对ChannelHandler进行更多的说明，但是目前你可以认为每个Channel-Handler的实例都类似于一种为了响应特定事件而被执行的回调。

Netty提供了大量预定义的可以开箱即用的ChannelHandler实现，包括用于各种协议（如HTTP和SSL/TLS）的ChannelHandler。在内部，ChannelHandler自己也使用了事件和Future，使得它们也成为了你的应用程序将使用的相同抽象的消费者。

### 1.3.5　把它们放在一起

在本章中，我们介绍了Netty实现高性能网络编程的方式，以及它的实现中的一些主要的组件。让我们大体回顾一下我们讨论过的内容吧。

#### 1．Future、回调和ChannelHandler

Netty的异步编程模型是建立在Future和回调的概念之上的， 而将事件派发到ChannelHandler的方法则发生在更深的层次上。结合在一起，这些元素就提供了一个处理环境，使你的应用程序逻辑可以独立于任何网络操作相关的顾虑而独立地演变。这也是Netty的设计方式的一个关键目标。

拦截操作以及高速地转换入站数据和出站数据，都只需要你提供回调或者利用操作所返回的Future。这使得链接操作变得既简单又高效，并且促进了可重用的通用代码的编写。

#### 2．选择器、事件和EventLoop

Netty通过触发事件将Selector从应用程序中抽象出来，消除了所有本来将需要手动编写的派发代码。在内部，将会为每个Channel分配一个EventLoop，用以处理所有事件，包括：

* 注册感兴趣的事件；
* 将事件派发给ChannelHandler；
* 安排进一步的动作。

EventLoop本身只由一个线程驱动，其处理了一个Channel的所有I/O事件，并且在该EventLoop的整个生命周期内都不会改变。这个简单而强大的设计消除了你可能有的在你的ChannelHandler中需要进行同步的任何顾虑，因此，你可以专注于提供正确的逻辑，用来在有感兴趣的数据要处理的时候执行。如同我们在详细探讨Netty的线程模型时将会看到的，该API是简单而紧凑的。

## 1.4　小结

在这一章中，我们介绍了Netty框架的背景知识，包括Java网络编程API的演变过程，阻塞和非阻塞网络操作之间的区别，以及异步I/O在高容量、高性能的网络编程中的优势。

然后，我们概述了Netty的特性、设计和优点，其中包括Netty异步模型的底层机制，包括回调、Future以及它们的结合使用。我们还谈到了事件是如何产生的以及如何拦截和处理它们。

在本书接下来的部分，我们将更加深入地探讨如何利用这些丰富的工具集来满足自己的应用程序的特定需求。

在下一章中，我们将要深入地探讨Netty的API以及编程模型的基础知识，而你则将编写你的第一款客户端和服务器应用程序。

**第2章　你的第一款Netty应用程序**

**本章主要内容**

* 设置开发环境
* 编写Echo服务器和客户端
* 构建并测试应用程序

在本章中，我们将展示如何构建一个基于Netty的客户端和服务器。应用程序很简单：客户端将消息发送给服务器，而服务器再将消息回送给客户端。但是这个练习很重要，原因有两个。

首先，它会提供一个测试台，用于设置和验证你的开发工具和环境，如果你打算通过对本书的示例代码的练习来为自己将来的开发工作做准备，那么它将是必不可少的。

其次，你将获得关于Netty的一个关键方面的实践经验，即在前一章中提到过的：通过ChannelHandler来构建应用程序的逻辑。这能让你对在第3章中开始的对Netty API的深入学习做好准备。

**2.1　设置开发环境**

要编译和运行本书的示例，只需要JDK和Apache Maven这两样工具，它们都是可以免费下载的。

我们将假设，你想要捣鼓示例代码，并且想很快就开始编写自己的代码。虽然你可以使用纯文本编辑器，但是我们仍然强烈地建议你使用用于Java的集成开发环境（IDE）。

**2.1.1　获取并安装Java开发工具包**

你的操作系统可能已经安装了JDK。为了找到答案，可以在命令行输入：

javac -version

如果得到的是javac 1.7……或者1.8……，则说明已经设置好了并且可以略过此步[[1]](http://www.epubit.com.cn/book/onlinechapter/52872#anchor21)。

否则，请从http://java.com/en/download/manual.jsp处获取JDK第8版。请留心，需要下载的是JDK，而不是Java运行时环境（JRE），其只可以运行Java应用程序，但是不能够编译它们。该网站为每个平台都提供了可执行的安装程序。如果需要安装说明，可以在同一个网站上找到相关的信息。

建议执行以下操作：

* 将环境变量JAVA\_HOME设置为你的JDK安装位置（在Windows上，默认值将类似于C:\Program Files\Java\jdk1.8.0\_121）；
* 将%JAVA\_HOME%\bin（在Linux上为${JAVA\_HOME}/bin）添加到你的执行路径。

**2.1.2　下载并安装IDE**

下面是使用最广泛的Java IDE，都可以免费获取：

* Eclipse—— www.eclipse.org；
* NetBeans—— www.netbeans.org；
* Intellij IDEA Community Edition—— www.jetbrains.com。

所有这3种对我们将使用的构建工具Apache Maven都拥有完整的支持。NetBeans和Intellij IDEA都通过可执行的安装程序进行分发。Eclipse通常使用Zip归档文件进行分发，当然也有一些自定义的版本包含了自安装程序。

**2.1.3　下载和安装Apache Maven**

即使你已经熟悉Maven了，我们仍然建议你至少大致浏览一下这一节。

Maven是一款广泛使用的由Apache软件基金会（ASF）开发的构建管理工具。Netty项目以及本书的示例都使用了它。构建和运行这些示例并不需要你成为一个Maven专家，但是如果你想要对其进行扩展，我们推荐你阅读附录中的Maven简介。

你需要安装Maven吗

Eclipse和NetBeans[[2]](http://www.epubit.com.cn/book/onlinechapter/52872#anchor22)自带了一个内置的Maven安装包，对于我们的目的来说开箱即可工作得良好。如果你将要在一个拥有它自己的Maven存储库的环境中工作，那么你的配置管理员可能就有一个预先配置好的能配合它使用的Maven安装包。

在本书中文版出版时，Maven 的最新版本是3.3.9。你可以从http://maven.apache.org/ download.cgi下载适用于你的操作系统的tar.gz或者zip归档文件[[3]](http://www.epubit.com.cn/book/onlinechapter/52872#anchor23)。安装很简单：将归档文件的所有内容解压到你所选择的任意的文件夹（我们将其称为<安装目录>）。这将创建目录<安装目录>\apache-maven-3.3.9。

和设置Java环境一样：

* 将环境变量M2\_HOME设置为指向<安装目录>\apache-maven-3.3.9；
* 将%M2\_HOME%\bin（或者在Linux上为${M2\_HOME}/bin）添加到你的执行路径。

这将使得你可以通过在命令行执行mvn.bat（或者mvn）来运行Maven。

**2.1.4　配置工具集**

如果你已经按照推荐设置好了环境变量JAVA\_HOME和M2\_HOME，那么你可能会发现，当你启动自己的IDE时，它已经发现了你的Java和Maven的安装位置。如果你需要进行手动配置，我们所列举的所有的IDE版本在Preferences或者Settings下都有设置这些变量的菜单项。相关的细节请查阅文档。

这就完成了开发环境的配置。在接下来的各节中，我们将介绍你要构建的第一个Netty应用程序的详细信息，同时我们将更加深入地了解该框架的API。之后，你就能使用刚刚设置好的工具来构建和运行Echo服务器和客户端了。

## 2.2　Netty客户端/服务器概览

图2-1从高层次上展示了一个你将要编写的Echo客户端和服务器应用程序。虽然你的主要关注点可能是编写基于Web的用于被浏览器访问的应用程序，但是通过同时实现客户端和服务器，你一定能更加全面地理解Netty的API。

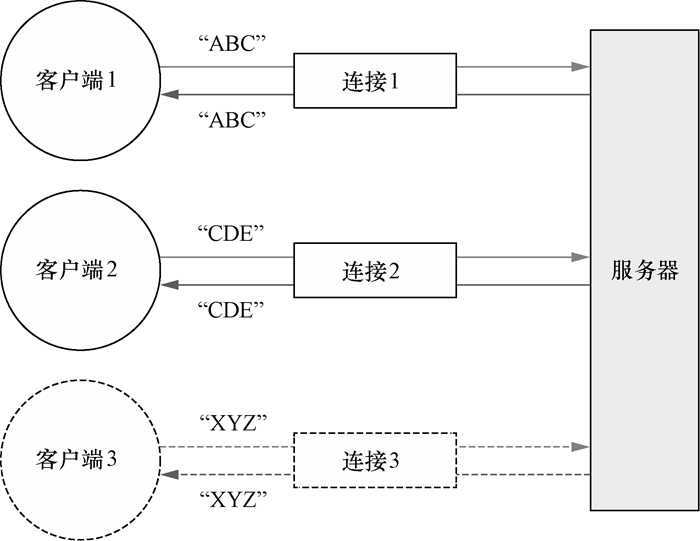


图2-1　Echo客户端和服务器

虽然我们已经谈及到了客户端，但是该图展示的是多个客户端同时连接到一台服务器。所能够支持的客户端数量，在理论上，仅受限于系统的可用资源（以及所使用的JDK版本可能会施加的限制）。

Echo客户端和服务器之间的交互是非常简单的；在客户端建立一个连接之后，它会向服务器发送一个或多个消息，反过来，服务器又会将每个消息回送给客户端。虽然它本身看起来好像用处不大，但它充分地体现了客户端/服务器系统中典型的请求-响应交互模式。

我们将从考察服务器端代码开始这个项目。

**2.3　编写Echo服务器**

所有的Netty服务器都需要以下两部分。

* **至少一个**ChannelHandler——该组件实现了服务器对从客户端接收的数据的处理，即它的业务逻辑。
* **引导**——这是配置服务器的启动代码。至少，它会将服务器绑定到它要监听连接请求的端口上。

在本小节的剩下部分，我们将描述Echo服务器的业务逻辑以及引导代码。

**2.3.1　ChannelHandler和业务逻辑**

在第1章中，我们介绍了Future和回调，并且阐述了它们在事件驱动设计中的应用。我们还讨论了ChannelHandler，它是一个接口族的父接口，它的实现负责接收并响应事件通知。在Netty应用程序中，所有的数据处理逻辑都包含在这些核心抽象的实现中。

因为你的Echo服务器会响应传入的消息，所以它需要实现ChannelInboundHandler接口，用来定义响应**入站**事件的方法。这个简单的应用程序只需要用到少量的这些方法，所以继承Channel-InboundHandlerAdapter类也就足够了，它提供了ChannelInboundHandler的默认实现。

我们感兴趣的方法是：

* channelRead()——对于每个传入的消息都要调用；
* channelReadComplete()——通知ChannelInboundHandler最后一次对channel-Read()的调用是当前批量读取中的最后一条消息；
* exceptionCaught()——在读取操作期间，有异常抛出时会调用。

该Echo服务器的ChannelHandler实现是EchoServerHandler，如代码清单2-1所示。

代码清单2-1　**EchoServerHandler**

@Sharable　　⇽---　标示一个Channel- Handler可以被多个Channel安全地共享

public class EchoServerHandler extends ChannelInboundHandlerAdapter {

@Override

public void channelRead(ChannelHandlerContext ctx, Object msg) {

ByteBuf in = (ByteBuf) msg;

System.out.println(

"Server received: " + in.toString(CharsetUtil.UTF\_8)); ⇽--- 将消息记录到控制台

ctx.write(in); 　　⇽---　将接收到的消息写给发送者，而不冲刷出站消息

}

@Override

public void channelReadComplete(ChannelHandlerContext ctx) {

ctx.writeAndFlush(Unpooled.EMPTY\_BUFFER)

.addListener(ChannelFutureListener.CLOSE); 　　⇽---　将未决消息冲刷到远程节点，并且关闭该Channel

}

@Override

public void exceptionCaught(ChannelHandlerContext ctx,

Throwable cause) {

*cause.printStackTrace(); ⇽--- 打印异常栈跟踪*

*ctx.close();*⇽---　关闭该Channel

}

}

ChannelInboundHandlerAdapter有一个直观的API，并且它的每个方法都可以被重写以挂钩到事件生命周期的恰当点上。因为需要处理所有接收到的数据，所以你重写了channelRead()方法。在这个服务器应用程序中，你将数据简单地回送给了远程节点。

重写exceptionCaught()方法允许你对Throwable的任何子类型做出反应，在这里你记录了异常并关闭了连接。虽然一个更加完善的应用程序也许会尝试从异常中恢复，但在这个场景下，只是通过简单地关闭连接来通知远程节点发生了错误。

如果不捕获异常，会发生什么呢

每个Channel都拥有一个与之相关联的ChannelPipeline，其持有一个ChannelHandler的实例链。在默认的情况下，ChannelHandler会把对它的方法的调用转发给链中的下一个Channel-Handler。因此，如果exceptionCaught()方法没有被该链中的某处实现，那么所接收的异常将会被传递到ChannelPipeline的尾端并被记录。为此，你的应用程序应该提供至少有一个实现了exceptionCaught()方法的ChannelHandler。（6.4节详细地讨论了异常处理）。

除了ChannelInboundHandlerAdapter之外，还有很多需要学习的ChannelHandler的子类型和实现，我们将在第6章和第7章中对它们进行详细的阐述。目前，请记住下面这些关键点：

* 针对不同类型的事件来调用ChannelHandler；
* 应用程序通过实现或者扩展ChannelHandler来挂钩到事件的生命周期，并且提供自定义的应用程序逻辑；
* 在架构上，ChannelHandler有助于保持业务逻辑与网络处理代码的分离。这简化了开发过程，因为代码必须不断地演化以响应不断变化的需求。

**2.3.2　引导服务器**

在讨论过由EchoServerHandler实现的核心业务逻辑之后，我们现在可以探讨引导服务器本身的过程了，具体涉及以下内容：

* 绑定到服务器将在其上监听并接受传入连接请求的端口；
* 配置Channel，以将有关的入站消息通知给EchoServerHandler实例。

传输

在这一节中，你将遇到术语传输。在网络协议的标准多层视图中，传输层提供了端到端的或者主机到主机的通信服务。

因特网通信是建立在TCP传输之上的。除了一些由Java NIO实现提供的服务器端性能增强之外，NIO传输大多数时候指的就是TCP传输。

我们将在第4章对传输进行详细的讨论。

代码清单2-2展示了EchoServer类的完整代码。

代码清单2-2　**Echo****Server**类

public class EchoServer {

private final int port;

public EchoServer(int port) {

this.port = port;

}

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

if (args.length != 1) {

System.err.println(

"Usage: " + EchoServer.class.getSimpleName() +

" ");

}

int port = Integer.parseInt(args[0]); 　　⇽---　设置端口值（如果端口参数的格式不正确，则抛出一个NumberFormatException）

new EchoServer(port).start(); ⇽--- 调用服务器的start()方法

}

public void start() throws Exception {

final EchoServerHandler serverHandler = new EchoServerHandler();

EventLoopGroup group = new NioEventLoopGroup(); ⇽--- ➊ 创建Event-LoopGroup

try {

ServerBootstrap b = new ServerBootstrap(); ⇽--- ❷ 创建Server-Bootstrap

b.group(group)

.channel(NioServerSocketChannel.class) 　　⇽---　 ❸ 指定所使用的NIO传输Channel

.localAddress(new InetSocketAddress(port)) 　　⇽---　 ❹ 使用指定的端口设置套接字地址

.childHandler(new ChannelInitializer(){ ⇽--- ❺添加一个EchoServer-

Handler到子Channel的ChannelPipeline

@Override

public void initChannel(SocketChannel ch)

throws Exception {

ch.pipeline().addLast(serverHandler);[[4]](http://www.epubit.com.cn/book/onlinechapter/52872#anchor24) 　　⇽---　 EchoServerHandler被标注为@Shareable，所以我们可以总是使用同样的实例

}

});

ChannelFuture f = b.bind().sync(); ⇽--- ❻ 异步地绑定服务器；调用sync()方法阻塞等待直到绑定完成

f.channel().closeFuture().sync();　　⇽---　 ❼ 获取Channel的CloseFuture，并且阻塞当前线程直到它完成

} finally {

group.shutdownGracefully().sync(); ⇽--- ❽ 关闭EventLoopGroup，释放所有的资源

}

}

}

在➋处，你创建了一个ServerBootstrap实例。因为你正在使用的是NIO传输，所以你指定了NioEventLoopGroup➊来接受和处理新的连接，并且将Channel的类型指定为NioServer-SocketChannel➌。在此之后，你将本地地址设置为一个具有选定端口的InetSocket-Address➍。服务器将绑定到这个地址以监听新的连接请求。

在➎处，你使用了一个特殊的类——ChannelInitializer。这是关键。当一个新的连接被接受时，一个新的子Channel将会被创建，而ChannelInitializer将会把一个你的EchoServerHandler的实例添加到该Channel的ChannelPipeline中。正如我们之前所解释的，这个ChannelHandler将会收到有关入站消息的通知。

虽然NIO是可伸缩的，但是其适当的尤其是关于多线程处理的配置并不简单。Netty的设计封装了大部分的复杂性，而且我们将在第3章中对相关的抽象（EventLoopGroup、Socket-Channel和ChannelInitializer）进行详细的讨论。

接下来你绑定了服务器➏，并等待绑定完成。（对sync()方法的调用将导致当前Thread阻塞，一直到绑定操作完成为止）。在➐处，该应用程序将会阻塞等待直到服务器的Channel关闭（因为你在Channel的Close Future上调用了sync()方法）。然后，你将可以关闭EventLoopGroup，并释放所有的资源，包括所有被创建的线程➑。

这个示例使用了NIO，因为得益于它的可扩展性和彻底的异步性，它是目前使用最广泛的传输。但是也可以使用一个不同的传输实现。如果你想要在自己的服务器中使用OIO传输，将需要指定OioServerSocketChannel和OioEventLoopGroup。我们将在第4章中对传输进行更加详细的探讨。

与此同时，让我们回顾一下你刚完成的服务器实现中的重要步骤。下面这些是服务器的主要代码组件：

* EchoServerHandler实现了业务逻辑；
* main()方法引导了服务器；

引导过程中所需要的步骤如下：

* 创建一个ServerBootstrap的实例以引导和绑定服务器；
* 创建并分配一个NioEventLoopGroup实例以进行事件的处理，如接受新连接以及读/写数据；
* 指定服务器绑定的本地的InetSocketAddress；
* 使用一个EchoServerHandler的实例初始化每一个新的Channel；
* 调用ServerBootstrap.bind()方法以绑定服务器。

在这个时候，服务器已经初始化，并且已经就绪能被使用了。在下一节中，我们将探讨对应的客户端应用程序的代码。

**2.4　编写Echo客户端**

Echo客户端将会：

（1）连接到服务器；

（2）发送一个或者多个消息；

（3）对于每个消息，等待并接收从服务器发回的相同的消息；

（4）关闭连接。

编写客户端所涉及的两个主要代码部分也是业务逻辑和引导，和你在服务器中看到的一样。

**2.4.1　通过ChannelHandler实现客户端逻辑**

如同服务器，客户端将拥有一个用来处理数据的ChannelInboundHandler。在这个场景下，你将扩展SimpleChannelInboundHandler类以处理所有必须的任务，如代码清单2-3所示。这要求重写下面的方法：

* channelActive()——在到服务器的连接已经建立之后将被调用；
* channelRead0()[[5]](http://www.epubit.com.cn/book/onlinechapter/52872#anchor25)——当从服务器接收到一条消息时被调用；
* exceptionCaught()——在处理过程中引发异常时被调用。

代码清单2-3　客户端的**ChannelHandler**

@Sharable ⇽--- 标记该类的实例可以被多个Channel共享

public class EchoClientHandler extends

SimpleChannelInboundHandler<ByteBuf> {

@Override

public void channelActive(ChannelHandlerContext ctx) {

ctx.writeAndFlush(Unpooled.copiedBuffer("Netty rocks!", ⇽--- 当被通知Channel是活跃的时候，发送一条消息

CharsetUtil.UTF\_8));

}

@Override

public void channelRead0(ChannelHandlerContext ctx, ByteBuf in) {

System.out.println( ⇽--- 记录已接收消息的转储

"Client received: " + in.toString(CharsetUtil.UTF\_8));

}

@Override

public void exceptionCaught(ChannelHandlerContext ctx, ⇽--- 在发生异常时，记录错误并关闭Channel

Throwable cause) {

cause.printStackTrace();

ctx.close();

}

}

首先，你重写了channelActive()方法，其将在一个连接建立时被调用。这确保了数据将会被尽可能快地写入服务器，其在这个场景下是一个编码了字符串"Netty rocks!"的字节缓冲区。

接下来，你重写了channelRead0()方法。每当接收数据时，都会调用这个方法。需要注意的是，由服务器发送的消息可能会被分块接收。也就是说，如果服务器发送了5字节，那么不能保证这5字节会被一次性接收。即使是对于这么少量的数据，channelRead0()方法也可能会被调用两次，第一次使用一个持有3字节的ByteBuf（Netty的字节容器），第二次使用一个持有2字节的ByteBuf。作为一个面向流的协议，TCP保证了字节数组将会按照服务器发送它们的顺序被接收。

重写的第三个方法是exceptionCaught()。如同在EchoServerHandler（见代码清单2-2）中所示，记录Throwable，关闭Channel，在这个场景下，终止到服务器的连接。

SimpleChannelInboundHandler与ChannelInboundHandler

你可能会想：为什么我们在客户端使用的是SimpleChannelInboundHandler，而不是在Echo- ServerHandler中所使用的ChannelInboundHandlerAdapter呢？这和两个因素的相互作用有关：业务逻辑如何处理消息以及Netty如何管理资源。

在客户端，当channelRead0()方法完成时，你已经有了传入消息，并且已经处理完它了。当该方法返回时，SimpleChannelInboundHandler负责释放指向保存该消息的ByteBuf的内存引用。

在EchoServerHandler中，你仍然需要将传入消息回送给发送者，而write()操作是异步的，直到channelRead()方法返回后可能仍然没有完成（如代码清单2-1所示）。为此，EchoServerHandler扩展了ChannelInboundHandlerAdapter，其在这个时间点上不会释放消息。

消息在EchoServerHandler的channelReadComplete()方法中，当writeAndFlush()方法被调用时被释放（见代码清单2-1）。

第5章和第6章将对消息的资源管理进行详细的介绍。

**2.4.2　引导客户端**

如同将在代码清单2-4中所看到的，引导客户端类似于引导服务器，不同的是，客户端是使用主机和端口参数来连接远程地址，也就是这里的Echo服务器的地址，而不是绑定到一个一直被监听的端口。

代码清单2-4　客户端的主类

public class EchoClient {

private final String host;

private final int port;

public EchoClient(String host, int port) {

this.host = host;

this.port = port;

}

public void start() throws Exception {

EventLoopGroup group = new NioEventLoopGroup();

try { ⇽--- 创建Bootstrap

Bootstrap b = new Bootstrap(); ⇽--- 指定EventLoopGroup以处理客户端事件；需要适用于NIO的实现

b.group(group)

.channel(NioSocketChannel.class) ⇽--- 适用于NIO传输的Channel类型

.remoteAddress(new InetSocketAddress(host, port)) ⇽--- 设置服务器的InetSocketAddr-ess

![](/api/storage/getbykey/screenshow?key=17043add7e9c14a5d3f7) .handler(new ChannelInitializer<SocketChannel>() { ⇽--- 在创建Channel时，向ChannelPipeline中添加一个Echo-ClientHandler实例

@Override

public void initChannel(SocketChannel ch)

throws Exception {

ch.pipeline().addLast(

new EchoClientHandler());

}

});

ChannelFuture f = b.connect().sync(); ⇽--- 连接到远程节点，阻塞等待直到连接完成

f.channel().closeFuture().sync(); ⇽--- 阻塞，直到Channel关闭

} finally {

group.shutdownGracefully().sync(); ⇽--- 关闭线程池并且释放所有的资源

}

}

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

if (args.length != 2) {

System.err.println(

"Usage: " + EchoClient.class.getSimpleName() +

" <host> <port>");

return;

}

String host = args[0];

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

new EchoClient(host, port).start();

}

}

和之前一样，使用了NIO传输。注意，你可以在客户端和服务器上分别使用不同的传输。例如，在服务器端使用NIO传输，而在客户端使用OIO传输。在第4章，我们将探讨影响你选择适用于特定用例的特定传输的各种因素和场景。

让我们回顾一下这一节中所介绍的要点：

* 为初始化客户端，创建了一个Bootstrap实例；
* 为进行事件处理分配了一个NioEventLoopGroup实例，其中事件处理包括创建新的连接以及处理入站和出站数据；
* 为服务器连接创建了一个InetSocketAddress实例；
* 当连接被建立时，一个EchoClientHandler实例会被安装到（该Channel的）ChannelPipeline中；
* 在一切都设置完成后，调用Bootstrap.connect()方法连接到远程节点；

完成了客户端，你便可以着手构建并测试该系统了。

**2.5　构建和运行Echo服务器和客户端**

在这一节中，我们将介绍编译和运行Echo服务器和客户端所需的所有步骤。

Echo客户端/服务器的Maven工程

这本书的附录使用Echo客户端/服务器工程的配置，详细地解释了多模块Maven工程是如何组织的。这部分内容对于构建和运行该应用程序来说并不是必读的，之所以推荐阅读这部分内容，是因为它能帮助你更好地理解本书的示例以及Netty项目本身。

**2.5.1　运行构建**

要构建Echo客户端和服务器，请进入到代码示例根目录下的chapter2目录执行以下命令：

mvn clean package

这将产生非常类似于代码清单2-5所示的输出（我们已经编辑忽略了几个构建过程中的非必要步骤）。

代码清单2-5　构建Echo客户端和服务器

[INFO] Scanning for projects...

[INFO] -------------------------------------------------------------------

[INFO] Reactor Build Order:

[INFO]

[INFO] Chapter 2. Your First Netty Application - Echo App

[INFO] Chapter 2. Echo Client

[INFO] Chapter 2. Echo Server

[INFO]

[INFO] -------------------------------------------------------------------

[INFO] Building Chapter 2. Your First Netty Application - 2.0-SNAPSHOT

[INFO] -------------------------------------------------------------------

[INFO]

[INFO] --- maven-clean-plugin:2.6.1:clean (default-clean) @ chapter2 ---

[INFO]

[INFO] -------------------------------------------------------------------

[INFO] Building Chapter 2. Echo Client 2.0-SNAPSHOT

[INFO] -------------------------------------------------------------------

[INFO]

[INFO] --- maven-clean-plugin:2.6.1:clean (default-clean)

@ echo-client ---

[INFO]

[INFO] --- maven-resources-plugin:2.6:resources (default-resources)

@ echo-client ---

[INFO] Using 'UTF-8' encoding to copy filtered resources.

[INFO] Copying 1 resource

[INFO]

[INFO] --- maven-compiler-plugin:3.3:compile (default-compile)

@ echo-client ---

[INFO] Changes detected - recompiling the module!

[INFO] Compiling 2 source files to

\netty-in-action\chapter2\Client\target\classes

[INFO]

[INFO] --- maven-resources-plugin:2.6:testResources (default-testResources)

@ echo-client ---

[INFO] Using 'UTF-8' encoding to copy filtered resources.

[INFO] skip non existing resourceDirectory

\netty-in-action\chapter2\Client\src\test\resources

[INFO]

[INFO] --- maven-compiler-plugin:3.3:testCompile (default-testCompile)

@ echo-client ---

[INFO] No sources to compile

[INFO]

[INFO] --- maven-surefire-plugin:2.18.1:test (default-test)

@ echo-client ---

[INFO] No tests to run.

[INFO]

[INFO] --- maven-jar-plugin:2.6:jar (default-jar) @ echo-client ---

[INFO] Building jar:

\netty-in-action\chapter2\Client\target\echo-client-2.0-SNAPSHOT.jar

[INFO]

[INFO] -------------------------------------------------------------------

[INFO] Building Chapter 2. Echo Server 2.0-SNAPSHOT

[INFO] -------------------------------------------------------------------

[INFO]

[INFO] --- maven-clean-plugin:2.6.1:clean (default-clean)

@ echo-server ---

[INFO]

[INFO] --- maven-resources-plugin:2.6:resources (default-resources)

@ echo-server ---

[INFO] Using 'UTF-8' encoding to copy filtered resources.

[INFO] Copying 1 resource

[INFO]

[INFO] --- maven-compiler-plugin:3.3:compile (default-compile)

@ echo-server ---

[INFO] Changes detected - recompiling the module!

[INFO] Compiling 2 source files to

\netty-in-action\chapter2\Server\target\classes

[INFO]

[INFO] --- maven-resources-plugin:2.6:testResources (default-testResources)

@ echo-server ---

[INFO] Using 'UTF-8' encoding to copy filtered resources.

[INFO] skip non existing resourceDirectory

\netty-in-action\chapter2\Server\src\test\resources

[INFO]

[INFO] --- maven-compiler-plugin:3.3:testCompile (default-testCompile)

@ echo-server ---

[INFO] No sources to compile

[INFO]

[INFO] --- maven-surefire-plugin:2.18.1:test (default-test)

@ echo-server ---

[INFO] No tests to run.

[INFO]

[INFO] --- maven-jar-plugin:2.6:jar (default-jar) @ echo-server ---

[INFO] Building jar:

\netty-in-action\chapter2\Server\target\echo-server-2.0-SNAPSHOT.jar

[INFO] -------------------------------------------------------------------

[INFO] Reactor Summary:

[INFO]

[INFO] Chapter 2. Your First Netty Application ... SUCCESS [ 0.134 s]

[INFO] Chapter 2. Echo Client .................... SUCCESS [ 1.509 s]

[INFO] Chapter 2. Echo Ser........................ SUCCESS [ 0.139 s]

[INFO] -------------------------------------------------------------------

[INFO] BUILD SUCCESS

[INFO] -------------------------------------------------------------------

[INFO] Total time: 1.886 s

[INFO] Finished at: 2015-11-18T17:14:10-05:00

[INFO] Final Memory: 18M/216M

[INFO] -------------------------------------------------------------------

下面是前面的构建日志中记录的主要步骤：

* Maven确定了构建顺序：首先是父pom.xml，然后是各个模块（子工程）；
* 如果在用户的本地存储库中没有找到Netty构件，Maven将从公共的Maven存储库中下载它们（此处未显示）；
* 运行了构建生命周期中的clean和compile阶段；
* 最后执行了maven-jar-plugin。

Maven Reactor的摘要显示所有的项目都已经被成功地构建。两个子工程的目标目录的文件列表现在应该类似于代码清单2-6。

代码清单2-6　构建的构件列表

Directory of nia\chapter2\Client\target

03/16/2015 09:45 PM <DIR> classes

03/16/2015 09:45 PM 5,614 echo-client-1.0-SNAPSHOT.jar

03/16/2015 09:45 PM <DIR> generated-sources

03/16/2015 09:45 PM <DIR> maven-archiver

03/16/2015 09:45 PM <DIR> maven-status

Directory of nia\chapter2\Server/target

03/16/2015 09:45 PM <DIR> classes

03/16/2015 09:45 PM 5,629 echo-server-1.0-SNAPSHOT.jar

03/16/2015 09:45 PM <DIR> generated-sources

03/16/2015 09:45 PM <DIR> maven-archiver

03/16/2015 09:45 PM <DIR> maven-status

**2.5.2　运行Echo服务器和客户端**

要运行这些应用程序组件，可以直接使用Java命令。但是在POM文件中，已经为你配置好了exec-maven-plugin来做这个（参见附录以获取详细信息）。

并排打开两个控制台窗口，一个进到chapter2\Server目录中，另外一个进到chapter2\Client目录中。

在服务器的控制台中执行这个命令：

mvn exec:java

应该会看到类似于下面的内容：

[INFO] Scanning for projects...

[INFO]

[INFO] ----------------------------------------------------------------------

[INFO] Building Echo Server 1.0-SNAPSHOT

[INFO] ----------------------------------------------------------------------

[INFO]

[INFO] >>> exec-maven-plugin:1.2.1:java (default-cli) >

validate @ echo-server >>>

[INFO]

[INFO] <<< exec-maven-plugin:1.2.1:java (default-cli) <

validate @ echo-server <<<

[INFO]

[INFO] --- exec-maven-plugin:1.2.1:java (default-cli) @ echo-server ---

nia.chapter2.echoserver.EchoServer

started and listening for connections on /0:0:0:0:0:0:0:0:9999

服务器现在已经启动并准备好接受连接。现在在客户端的控制台中执行同样的命令：

mvn exec:java

应该会看到下面的内容：

[INFO] Scanning for projects...

[INFO]

[INFO] -------------------------------------------------------------------

[INFO] Building Echo Client 1.0-SNAPSHOT

[INFO] -------------------------------------------------------------------

[INFO]

[INFO] >>> exec-maven-plugin:1.2.1:java (default-cli) >

validate @ echo-client >>>

[INFO]

[INFO] <<< exec-maven-plugin:1.2.1:java (default-cli) <

validate @ echo-client <<<

[INFO]

[INFO] --- exec-maven-plugin:1.2.1:java (default-cli) @ echo-client ---

Client received: Netty rocks!

[INFO] -------------------------------------------------------------------

[INFO] BUILD SUCCESS

[INFO] -------------------------------------------------------------------

[INFO] Total time: 2.833 s

[INFO] Finished at: 2015-03-16T22:03:54-04:00

[INFO] Final Memory: 10M/309M

[INFO] -------------------------------------------------------------------

同时在服务器的控制台中，应该会看到这个：

Server received: Netty rocks!

每次运行客户端时，在服务器的控制台中你都能看到这条日志语句。

下面是发生的事：

（1）一旦客户端建立连接，它就发送它的消息——Netty rocks!；

（2）服务器报告接收到的消息，并将其回送给客户端；

（3）客户端报告返回的消息并退出。

你所看到的都是预期的行为，现在让我们看看故障是如何被处理的。服务器应该还在运行，所以在服务器的控制台中按下Ctrl+C来停止该进程。一旦它停止，就再次使用下面的命令启动客户端：

mvn exec:java

代码清单2-7展示了你应该会从客户端的控制台中看到的当它不能连接到服务器时的输出。

代码清单2-7　Echo客户端的异常处理

[INFO] Scanning for projects...

[INFO]

[INFO] --------------------------------------------------------------------

[INFO] Building Echo Client 1.0-SNAPSHOT

[INFO] --------------------------------------------------------------------

[INFO]

[INFO] >>> exec-maven-plugin:1.2.1:java (default-cli) >

validate @ echo-client >>>

[INFO]

[INFO] <<< exec-maven-plugin:1.2.1:java (default-cli) <

validate @ echo-client <<<

[INFO]

[INFO] --- exec-maven-plugin:1.2.1:java (default-cli) @ echo-client ---

[WARNING]

java.lang.reflect.InvocationTargetException

at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)

. . .

Caused by: java.net.ConnectException: Connection refused:

no further information: localhost/127.0.0.1:9999

at sun.nio.ch.SocketChannelImpl.checkConnect(Native Method)

at sun.nio.ch.SocketChannelImpl

.finishConnect(SocketChannelImpl.java:739)

at io.netty.channel.socket.nio.NioSocketChannel

.doFinishConnect(NioSocketChannel.java:208)

at io.netty.channel.nio

.AbstractNioChannel$AbstractNioUnsafe

.finishConnect(AbstractNioChannel.java:281)

at io.netty.channel.nio.NioEventLoop

.processSelectedKey(NioEventLoop.java:528)

at io.netty.channel.nio.NioEventLoop.

processSelectedKeysOptimized(NioEventLoop.java:468)

at io.netty.channel.nio.NioEventLoop

.processSelectedKeys(NioEventLoop.java:382)

at io.netty.channel.nio.NioEventLoop

.run(NioEventLoop.java:354)

at io.netty.util.concurrent.SingleThreadEventExecutor$2

.run(SingleThreadEventExecutor.java:116)

at io.netty.util.concurrent.DefaultThreadFactory

$DefaultRunnableDecorator.run(DefaultThreadFactory.java:137)

. . .

[INFO] --------------------------------------------------------------------

[INFO] BUILD FAILURE

[INFO] --------------------------------------------------------------------

[INFO] Total time: 3.801 s

[INFO] Finished at: 2015-03-16T22:11:16-04:00

[INFO] Final Memory: 10M/309M

[INFO] --------------------------------------------------------------------

[ERROR] Failed to execute goal org.codehaus.mojo:

exec-maven-plugin:1.2.1:java (default-cli) on project echo-client:

An exception occured while executing the Java class. null:

InvocationTargetException: Connection refused:

no further information: localhost/127.0.0.1:9999 -> [Help 1]

发生了什么？客户端试图连接服务器，其预期运行在localhost:9999上。但是连接失败了（和预期的一样），因为服务器在这之前就已经停止了，所以在客户端导致了一个java.net.ConnectException。这个异常触发了EchoClientHandler的exceptionCaught()方法，打印出了栈跟踪并关闭了Channel（见代码清单2-3）。

**2.6　小结**

在本章中，你设置好了开发环境，并且构建和运行了你的第一款Netty客户端和服务器。虽然这只是一个简单的应用程序，但是它可以伸缩到支持数千个并发连接——每秒可以比普通的基于套接字的Java应用程序处理多得多的消息。

在接下来的几章中，你将看到更多关于Netty如何简化可伸缩性和并发性的例子。我们也将更加深入地了解Netty对于关注点分离的架构原则的支持。通过提供正确的抽象来解耦业务逻辑和网络编程逻辑，Netty使得可以很容易地跟上快速演化的需求，而又不危及系统的稳定性。

在下一章中，我们将提供对Netty体系架构的概述。这将为你在后续的章节中对Netty的内部进行深入而全面的学习提供上下文。