# 第1章 Netty——异步和事件驱动

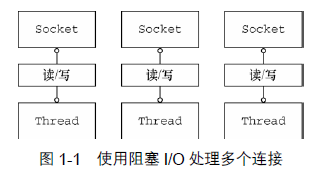
Netty 是一款异步的事件驱动的网络应用程序框架，支持快速地开发可维护的高性能的面向协议的服务器和客户端。

## Java 网络编程

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。

|  |
| --- |
| **public class** BlockingIoExample {  **public void** serve(**int** portNumber) **throws** IOException {  *// 创建一个新的ServerSocket，用以监听指定端口上的连接请求* ServerSocket serverSocket = **new** ServerSocket(portNumber);  *// 对accept()方法的调用将被阻塞，直到一个连接建立* Socket clientSocket = serverSocket.accept();   *// 流对象都派生于该套接字的流对象* BufferedReader in = **new** BufferedReader(**new** InputStreamReader(clientSocket.getInputStream()));  PrintWriter out = **new** PrintWriter(clientSocket.getOutputStream(), **true**);  String request;  String response;   *// 处理循环开始* **while** ((request = in.readLine()) != **null**) {  *// 如果客户端发送了“Done”，则退出处理循环* **if** (**"Done"**.endsWith(request)) {  **break**;  }  *// 请求被传递给服务器的处理方法* response = processRequest(request);  *// 服务器的响应被发送给了客户端* out.println(response);  }  }   **private** String processRequest(String request) {  **return "Processed"**;  } } |

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。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



### Java NIO

除了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；
* 可以使用操作系统的事件通知API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。

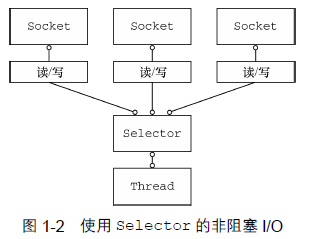
Java 对于非阻塞I/O 的支持是在2002 年引入的，位于JDK 1.4 的java.nio 包中。

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。

### 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。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：

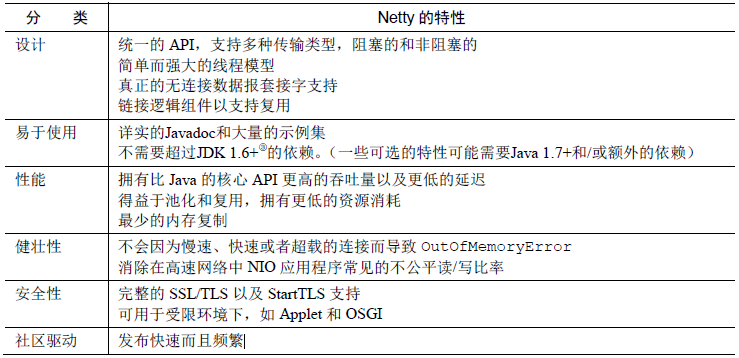
* 使用较少的线程便可以处理许多连接，因此也减少了内存管理和上下文切换所带来开销；
* 当没有I/O 操作需要处理的时候，线程也可以被用于其他任务。

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。

## Netty 简介

在网络编程领域，Netty是Java的卓越框架。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

表1-1 Netty 的特性总结



### 1.2.1 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。”**

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。

## 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。

### 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。

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。

### 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。

|  |
| --- |
| **public class** ConnectHandler **extends** ChannelInboundHandlerAdapter {  @Override  **public void** channelActive(ChannelHandlerContext ctx) **throws** Exception {  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。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**  **public class** ConnectExample {  **private static final** Channel ***CHANNEL\_FROM\_SOMEWHERE*** = **new** NioServerSocketChannel();   **public static void** connect() {  Channel channel = ***CHANNEL\_FROM\_SOMEWHERE***; *// 指向某个位置   // 不会阻塞，异步地连接到远程节点* ChannelFuture future = channel.connect(**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。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**  **public class** ConnectExample {  **private static final** Channel ***CHANNEL\_FROM\_SOMEWHERE*** = **new** NioServerSocketChannel();   **public static void** connect() {  Channel channel = ***CHANNEL\_FROM\_SOMEWHERE***; *// 指向某个位置   // 不会阻塞，异步地连接到远程节点* ChannelFuture future = channel.connect(**new** InetSocketAddress(**"192.168.0.1"**, **25**));  *// 注册一个ChannelFutureListener，以便在操作完成时获得通知* future.addListener(**new** ChannelFutureListener() {  @Override  **public void** operationComplete(ChannelFuture future) **throws** Exception {  *// 检查操作的状态* **if** (future.isSuccess()) {  *// 如果操作是成功的，则创建一个ByteBuf 以持有数据* ByteBuf buffer = Unpooled.*copiedBuffer*(**"Hello"**, Charset.*forName*(**"UTF-8"**));  *// 将数据异步地发送到远程节点。返回一个ChannelFuture* ChannelFuture wf = future.channel().write(buffer);  *// ...* } **else** {  *// 如果发生错误，则访问描述原因的Throwable* Throwable cause = future.cause();  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。

### 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。

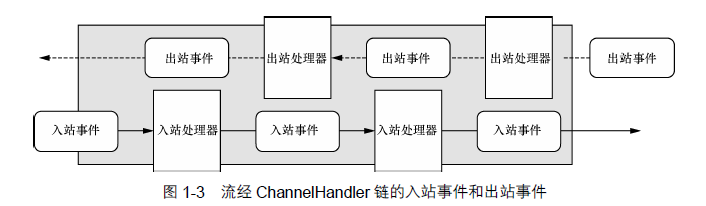
Netty 是一个网络编程框架，所以事件是按照它们与入站或出站数据流的相关性进行分类的。可能由入站数据或者相关的状态更改而触发的事件包括：

* 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。



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。

### 把它们放在一起

**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ChannelHandler 实现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。