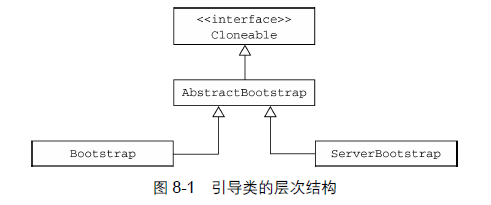
# 【第08章-引导】

引导一个应用程序是指对它进行配置，并使它运行起来的过程。Netty处理引导的方式使你的应用程序和网络层相隔离，无论它是客户端还是服务器

## 8.1 Bootstrap 类

引导类的层次结构包括一个抽象的父类和两个具体的引导子类，如图8-1 所示。



相对于将具体的引导类分别看作用于服务器和客户端的引导来说，记住它们的本意是用来支撑不同的应用程序的功能的将有所裨益。也就是说，**服务器致力于使用一个父Channel 来接受来自客户端的连接，并创建子Channel 以用于它们之间的通信；而客户端将最可能只需要一个单独的、没有父Channel 的Channel 来用于所有的网络交互。**（正如同我们将要看到的，这也适用于无连接的传输协议，如UDP，因为它们并不是每个连接都需要一个单独的Channel。）

我们在前面的几章中学习的几个Netty 组件都参与了引导的过程，而且其中一些在客户端和服务器都有用到。两种应用程序类型之间通用的引导步骤由AbstractBootstrap 处理，而特定于客户端或者服务器的引导步骤则分别由Bootstrap 或ServerBootstrap 处理。

在本章中接下来的部分，我们将详细地探讨这两个类，首先从不那么复杂的Bootstrap 类开始。

**为什么引导类是Cloneable 的**

你有时可能会需要创建多个具有类似配置或者完全相同配置的Channel。为了支持这种模式而又不需要为每个Channel 都创建并配置一个新的引导类实例， AbstractBootstrap 被标记为了Cloneable，在一个已经配置完成的引导类实例上调用clone()方法将返回另一个可以立即使用的引导类实例。

注意，这种方式只会创建引导类实例的EventLoopGroup的一个浅拷贝，所以，后者将在所有克隆的Channel实例之间共享。这是可以接受的，因为通常这些克隆的Channel的生命周期都很短暂，一个典型的场景是——创建一个Channel以进行一次HTTP请求。

AbstractBootstrap 类的完整声明是：

public abstract class AbstractBootstrap<B extends AbstractBootstrap<B,C>,C extends Channel>

在这个签名中，子类型B 是其父类型的一个类型参数，因此可以返回到运行时实例的引用以支持方法的链式调用（也就是所谓的流式语法）。

其子类的声明如下：

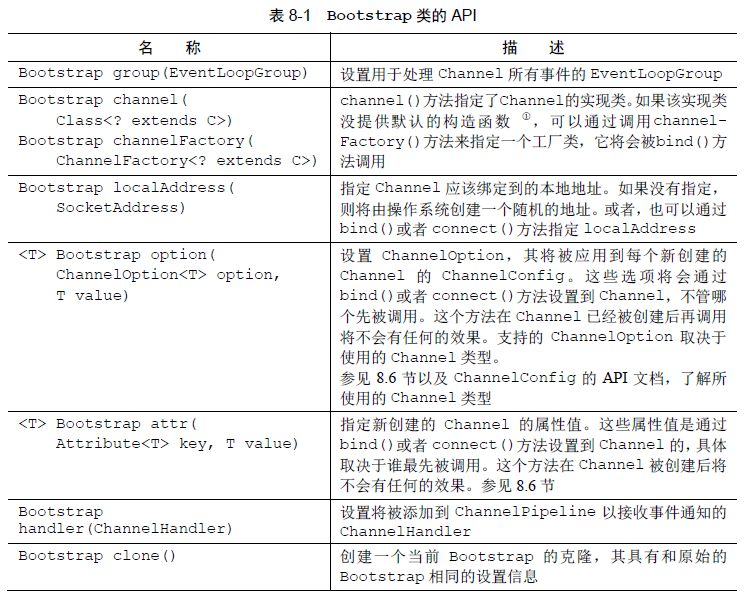
public class Bootstrap extends AbstractBootstrap<Bootstrap,Channel>

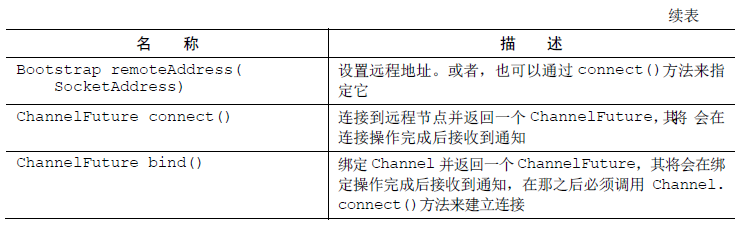
和

public class ServerBootstrap extends AbstractBootstrap<ServerBootstrap,ServerChannel>

## 8.2 引导客户端和无连接协议

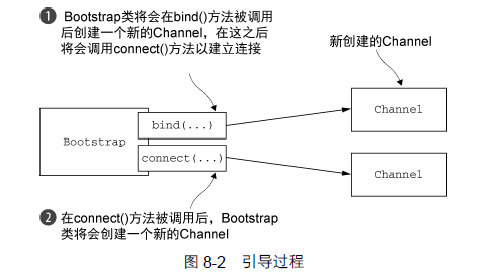
Bootstrap 类被用于客户端或者使用了无连接协议的应用程序中。表8-1 提供了该类的一个概览，其中许多方法都继承自AbstractBootstrap 类。





### 8.2.1 引导客户端

Bootstrap 类负责为客户端和使用无连接协议的应用程序创建Channel，如图8-2 所示

。

|  |
| --- |
| **// 代码清单8-1 引导一个客户端**  **public class** BootstrapClient {  **public static void** main(String args[]) {  BootstrapClient client = **new** BootstrapClient();  client.bootstrap();  } **public void** bootstrap() {  EventLoopGroup group = **new** NioEventLoopGroup();  *// 创建一个Bootstrap类的实例以创建和连接新的客户端Channel* Bootstrap bootstrap = **new** Bootstrap();  *// 设置EventLoopGroup，提供用于处理Channel事件的EventLoop* bootstrap.group(group)  .channel(NioSocketChannel.**class**)  .handler(**new** SimpleChannelInboundHandler<ByteBuf>() {  @Override  **protected void** channelRead0(  ChannelHandlerContext channelHandlerContext,  ByteBuf byteBuf) **throws** Exception {  System.***out***.println(**"Received data"**);  }  });  *// 连接到远程主机* ChannelFuture future = bootstrap.connect(  **new** InetSocketAddress(**"www.manning.com"**, **80**));  future.addListener(**new** ChannelFutureListener() {  @Override  **public void** operationComplete(ChannelFuture channelFuture) **throws** Exception {  **if** (channelFuture.isSuccess()) {  System.***out***.println(**"Connection established"**);  } **else** {  System.***err***.println(**"Connection attempt failed"**);  channelFuture.cause().printStackTrace();  }  }  });  } } |

这个示例使用了前面提到的流式语法；这些方法（除了connect()方法以外）将通过每次方法调用所返回的对Bootstrap 实例的引用链接在一起。

|  |
| --- |
| channel  ├───nio  │ NioEventLoopGroup  ├───oio  │ OioEventLoopGroup  └───socket  ├───nio  │ NioDatagramChannel  │ NioServerSocketChannel  │ NioSocketChannel  └───oio  OioDatagramChannel  OioServerSocketChannel  OioSocketChannel |

必须保持这种兼容性，不能混用具有不同前缀的组件，如NioEventLoopGroup 和OioSocketChannel。代码清单8-3 展示了试图这样做的一个例子。

|  |
| --- |
| **// 代码清单8-3 不兼容的Channel 和EventLoopGroup**  **public void** bootstrap() {  EventLoopGroup group = **new** NioEventLoopGroup();  *// 创建一个新的Bootstrap类的实例，以创建新的客户端Channel* Bootstrap bootstrap = **new** Bootstrap();  *// 指定一个适用于NIO 的EventLoopGroup 实现* bootstrap.group(group)  *// 指定一个适用于OIO 的Channel实现类* .channel(OioSocketChannel.**class**)  .handler(**new** SimpleChannelInboundHandler<ByteBuf>() {  @Override  **protected void** channelRead0(ChannelHandlerContext ctx, ByteBuf buf) **throws** Exception {  System.***out***.println(**"Received data"**);  }  });  *// 尝试连接到远程节点* ChannelFuture future = bootstrap.connect(  **new** InetSocketAddress(**"www.manning.com"**, **80**));  future.syncUninterruptibly(); } |

这段代码将会导致IllegalStateException，因为它混用了不兼容的传输。

Exception in thread "main" java.lang.IllegalStateException:

incompatible event loop type: io.netty.channel.nio.NioEventLoop at

io.netty.channel.AbstractChannel$AbstractUnsafe.register(

Abstrac tChannel.java:571)

关于IllegalStateException 的更多讨论

在引导的过程中，在调用bind()或者connect()方法之前，必须调用以下方法来设置所需的组件：

* group()；
* channel()或者channelFactory()；
* handler()。

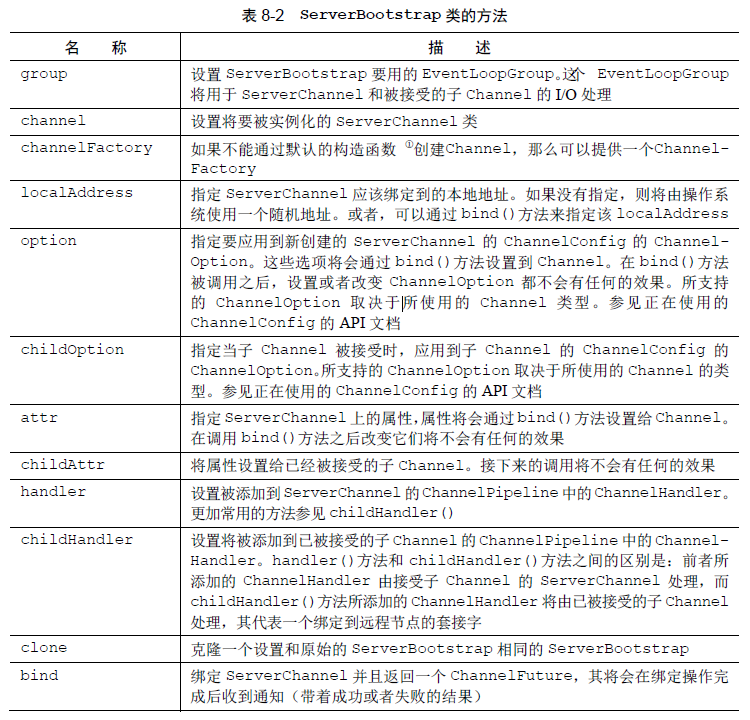
如果不这样做，则将会导致IllegalStateException。对handler()方法的调用尤其重要，因为它需要配置好ChannelPipeline。

## 8.3 引导服务器

我们将从ServerBootstrap API 的概要视图开始我们对服务器引导过程的概述。然后，我们将会探讨引导服务器过程中所涉及的几个步骤，以及几个相关的主题，包含从一个ServerChannel 的子Channel 中引导一个客户端这样的特殊情况。

### 8.3.1 ServerBootstrap 类

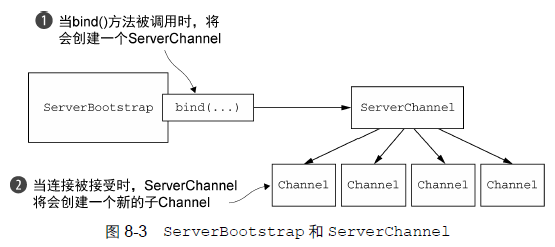
表8-2 列出了ServerBootstrap 类的方法。



### 8.3.2 引导服务器

你可能已经注意到了，表8-2 中列出了一些在表8-1 中不存在的方法：childHandler()、childAttr()和childOption()。这些调用支持特别用于服务器应用程序的操作。具体来说，ServerChannel 的实现负责创建子Channel，这些子Channel 代表了已被接受的连接。因此，负责引导ServerChannel 的ServerBootstrap 提供了这些方法，以简化将设置应用到已被接受的子Channel 的ChannelConfig 的任务。

图8-3 展示了ServerBootstrap 在bind()方法被调用时创建了一个ServerChannel，并且该ServerChannel 管理了多个子Channel。



代码清单8-4 中的代码实现了图8-3 中所展示的服务器的引导过程。

|  |
| --- |
| **// 代码清单8-4 引导服务器**  **public void** bootstrap() {  NioEventLoopGroup group = **new** NioEventLoopGroup();  *// 创建ServerBootstrap* ServerBootstrap bootstrap = **new** ServerBootstrap();  *// 设置EventLoopGroup，其提供了用于处理Channel 事件的EventLoop* bootstrap.group(group)  *// 指定要使用的Channel 实现* .channel(NioServerSocketChannel.**class**)  *// 设 置用于处理已被接受的子Channel的I/O及数据的ChannelInboundHandler* .childHandler(**new** SimpleChannelInboundHandler<ByteBuf>() {  @Override  **protected void** channelRead0(ChannelHandlerContext channelHandlerContext, ByteBuf byteBuf) **throws** Exception {  System.***out***.println(**"Received data"**);  }  });  ChannelFuture future = bootstrap.bind(**new** InetSocketAddress(**8080**));  future.addListener(**new** ChannelFutureListener() {  @Override  **public void** operationComplete(ChannelFuture channelFuture) **throws** Exception {  **if** (channelFuture.isSuccess()) {  System.***out***.println(**"Server bound"**);  } **else** {  System.***err***.println(**"Bind attempt failed"**);  channelFuture.cause().printStackTrace();  }  }  }); } |

我们在这一节中所讨论的主题以及所提出的解决方案都反映了**编写Netty 应用程序的一个一般准则：尽可能地重用EventLoop，以减少线程创建所带来的开销。**

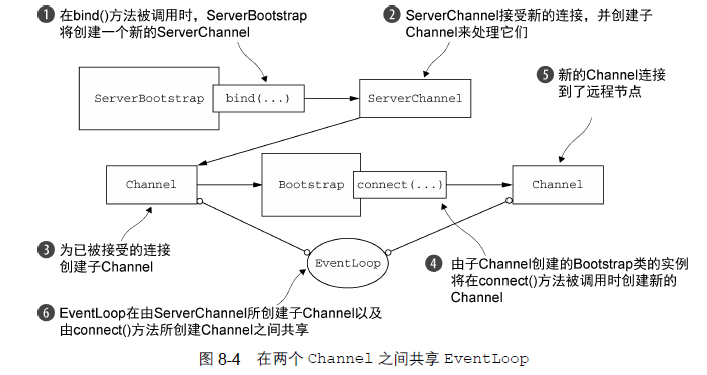
## 8.4 从Channel 引导客户端

假设你的服务器正在处理一个客户端的请求，这个请求需要它充当第三方系统的客户端。当一个应用程序（如一个代理服务器）必须要和组织现有的系统（如Web 服务或者数据库）集成时，就可能发生这种情况。在这种情况下，将需要从已经被接受的子Channel 中引导一个客户端Channel。

你可以按照8.2.1 节中所描述的方式创建新的Bootstrap 实例，但是这并不是最高效的解决方案，因为它将要求你为每个新创建的客户端Channel 定义另一个EventLoop。这会产生额外的线程，以及在已被接受的子Channel 和客户端Channel 之间交换数据时不可避免的上下文切换。

一个更好的解决方案是：通过将已被接受的子Channel 的EventLoop 传递给Bootstrap的group()方法来共享该EventLoop。因为分配给EventLoop 的所有Channel 都使用同一个线程，所以这避免了额外的线程创建，以及前面所提到的相关的上下文切换。这个共享的解决

方案如图8-4 所示。



实现EventLoop 共享涉及通过调用group()方法来设置EventLoop，如代码清单8-5 所示。

|  |
| --- |
| **// 代码清单8-5 引导服务器**  **public void** bootstrap() {  *// 创建ServerBootstrap 以创建ServerSocketChannel，并绑定它* ServerBootstrap bootstrap = **new** ServerBootstrap();  *// 设置EventLoopGroup，其将提供用以处理Channel 事件的EventLoop* bootstrap.group(**new** NioEventLoopGroup(), **new** NioEventLoopGroup())  *// 指定要使用的Channel 实现* .channel(NioServerSocketChannel.**class**)  *// 设置用于处理已被接受的子Channel 的I/O 和数据的ChannelInboundHandler* .childHandler(**new** SimpleChannelInboundHandler<ByteBuf>() {  ChannelFuture **connectFuture**;   @Override  **public void** channelActive(ChannelHandlerContext ctx)  **throws** Exception {  *// 创建一个Bootstrap类的实例以连接到远程主机* Bootstrap bootstrap = **new** Bootstrap();  *// 指定Channel的实现* bootstrap.channel(NioSocketChannel.**class**)  *// 为入站I/O 设置ChannelInboundHandler* .handler(**new** SimpleChannelInboundHandler<ByteBuf>() {  @Override  **protected void** channelRead0(ChannelHandlerContext ctx, ByteBuf in) **throws** Exception {  System.***out***.println(**"Received data"**);  }  });  *// 使用与分配给已被接受的子Channel 相同的EventLoop* bootstrap.group(ctx.channel().eventLoop());  *// 连接到远程节点* **connectFuture** = bootstrap.connect(**new** InetSocketAddress(**"www.manning.com"**, **80**));  }   @Override  **protected void** channelRead0(ChannelHandlerContext channelHandlerContext, ByteBuf byteBuf) **throws** Exception {  **if** (**connectFuture**.isDone()) {  *// 当连接完成时，执行一些数据操作（如代理）  // do something with the data* }  }  });  *// 通过配置好的ServerBootstrap绑定该ServerSocketChannel* ChannelFuture future = bootstrap.bind(**new** InetSocketAddress(**8080**));  future.addListener(**new** ChannelFutureListener() {  @Override  **public void** operationComplete(ChannelFuture channelFuture) **throws** Exception {  **if** (channelFuture.isSuccess()) {  System.***out***.println(**"Server bound"**);  } **else** {  System.***err***.println(**"Bind attempt failed"**);  channelFuture.cause().printStackTrace();  }  }  }); } |

## 8.5 在引导过程中添加多个ChannelHandler

在所有我们展示过的代码示例中，我们都在引导的过程中调用了handler()或者childHandler()方法来添加单个的ChannelHandler。这对于简单的应用程序来说可能已经足够了，但是它不能满足更加复杂的需求。例如，一个必须要支持多种协议的应用程序将会有很多的ChannelHandler，而不会是一个庞大而又笨重的类。

正如你经常所看到的一样，你可以根据需要，通过在ChannelPipeline 中将它们链接在一起来部署尽可能多的ChannelHandler。但是，如果在引导的过程中你只能设置一个ChannelHandler，那么你应该怎么做到这一点呢？

正是针对于这个用例，Netty 提供了一个特殊的ChannelInboundHandlerAdapter 子类：

public abstract class ChannelInitializer<C extends Channel> extends ChannelInboundHandlerAdapter

它定义了下面的方法：

protected abstract void initChannel(C ch) throws Exception;

这个方法提供了一种将多个ChannelHandler 添加到一个ChannelPipeline 中的简便方法。你只需要简单地向Bootstrap 或ServerBootstrap 的实例提供你的ChannelInitializer 实现即可，并且一旦Channel 被注册到了它的EventLoop 之后，就会调用你的initChannel()版本。在该方法返回之后，ChannelInitializer 的实例将会从ChannelPipeline 中移除它自己。

代码清单8-6 定义了ChannelInitializerImpl 类， 并通过ServerBootstrap 的childHandler()方法注册它，在大部分的场景下，如果你不需要使用只存在于SocketChannel 上的方法，使用ChannelInitializer<Channel>就可以了，否则你可以使用ChannelInitializer<SocketChannel>，其中SocketChannel扩展了Channel。。你可以看到，这个看似复杂的操作实际上是相当简单直接的。

|  |
| --- |
| **// 代码清单8-6 引导和使用ChannelInitializer**  **public void** bootstrap() **throws** InterruptedException {  *// 创建ServerBootstrap 以创建和绑定新的Channel* ServerBootstrap bootstrap = **new** ServerBootstrap();  *// 设置EventLoopGroup，其将提供用以处理Channel事件的EventLoop* bootstrap.group(  **new** NioEventLoopGroup(),  **new** NioEventLoopGroup()) *// 指定Channel的实现* .channel(NioServerSocketChannel.**class**)*//* .childHandler(**new** ChannelInitializerImpl()); *// 注册一个ChannelInitializerImpl 的实例来设置ChannelPipeline* ChannelFuture future = bootstrap.bind(**new** InetSocketAddress(**8080**)); *//绑定到地址* future.sync(); }  *// 用以设置ChannelPipeline 的自定义ChannelInitializerImpl 实现 // 在大部分的场景下，如果你不需要使用只存在于SocketChannel 上的方法， // 使用ChannelInitializer<Channel>就可以了，否则你可以使用 // ChannelInitializer<SocketChannel>，其中SocketChannel扩展了Channel。* **final class** ChannelInitializerImpl **extends** ChannelInitializer<Channel> {  @Override  **protected void** initChannel(Channel ch) **throws** Exception {  ChannelPipeline pipeline = ch.pipeline();  pipeline.addLast(**new** HttpClientCodec());  pipeline.addLast(**new** HttpObjectAggregator(Integer.***MAX\_VALUE***));   } } |

如果你的应用程序使用了多个ChannelHandler，请定义你自己的ChannelInitializer实现来将它们安装到ChannelPipeline 中。

## 8.6 使用Netty 的ChannelOption 和属性

在每个Channel 创建时都手动配置它可能会变得相当乏味。幸运的是，你不必这样做。相反，你可以使用option()方法来将ChannelOption 应用到引导。你所提供的值将会被自动应用到引导所创建的所有Channel。可用的ChannelOption 包括了底层连接的详细信息，如keep-alive 或者超时属性以及缓冲区设置。

Netty 应用程序通常与组织的专有软件集成在一起，而像Channel 这样的组件可能甚至会在正常的Netty 生命周期之外被使用。在某些常用的属性和数据不可用时，Netty 提供了AttributeMap 抽象（一个由Channel 和引导类提供的集合）以及AttributeKey<T>（一个用于插入和获取属性值的泛型类）。使用这些工具，便可以安全地将任何类型的数据项与客户端和服务器Channel（包含ServerChannel 的子Channel）相关联了。

例如，考虑一个用于跟踪用户和Channel 之间的关系的服务器应用程序。这可以通过将用户的ID 存储为Channel 的一个属性来完成。类似的技术可以被用来基于用户的ID 将消息路由给用户，或者关闭活动较少的Channel。

代码清单8-7 展示了可以如何使用ChannelOption 来配置Channel，以及如果使用属性来存储整型值。

|  |
| --- |
| **// 代码清单8-7 使用属性值**  **public void** bootstrap() {  *// 创建一个AttributeKey以标识该属性* **final** AttributeKey<Integer> id = AttributeKey.*newInstance*(**"ID"**);  *// 创建一个Bootstrap 类的实例以创建客户端Channel 并连接它们* Bootstrap bootstrap = **new** Bootstrap();  *// 设置EventLoopGroup，其提供了用以处理Channel事件的EventLoop* bootstrap.group(**new** NioEventLoopGroup())  .channel(NioSocketChannel.**class**) *// 指定Channel的实现  // 设置用以处理Channel 的I/O 以及数据的ChannelInboundHandler* .handler(**new** SimpleChannelInboundHandler<ByteBuf>() {  @Override  **public void** channelRegistered(ChannelHandlerContext ctx) **throws** Exception {  *// 使用AttributeKey 检索属性以及它的值* Integer idValue = ctx.channel().attr(id).get();  *// do something with the idValue* }   @Override  **protected void** channelRead0(ChannelHandlerContext channelHandlerContext, ByteBuf byteBuf) **throws** Exception {  System.***out***.println(**"Received data"**);  }  });  bootstrap.option(ChannelOption.***SO\_KEEPALIVE***, **true**)  *// 设置ChannelOption，其将在connect()或者bind()方法被调用时被设置到已经创建的Channel 上* .option(ChannelOption.***CONNECT\_TIMEOUT\_MILLIS***, **5000**);  *// 存储该id 属性* bootstrap.attr(id, **123456**);  ChannelFuture future = bootstrap.connect(  *// 使用配置好的Bootstrap实例连接到远程主机* **new** InetSocketAddress(**"www.manning.com"**, **80**));  future.syncUninterruptibly(); } |

## 8.7 引导DatagramChannel

前面的引导代码示例使用的都是基于TCP 协议的SocketChannel，但是Bootstrap 类也可以被用于无连接的协议。为此，Netty 提供了各种DatagramChannel 的实现。唯一区别就是，不再调用connect()方法，而是只调用bind()方法，如代码清单8-8 所示。

|  |
| --- |
| **// 代码清单8-8 使用Bootstrap 和DatagramChannel**  **public void** bootstrap() {  *// 创建一个Bootstrap 的实例以创建和绑定新的数据报Channel* Bootstrap bootstrap = **new** Bootstrap();  *// 设置EventLoopGroup，其提供了用以处理Channel 事件的EventLoop* bootstrap.group(**new** OioEventLoopGroup())  *// 指定Channel的实现* .channel(OioDatagramChannel.**class**)  *// 设置用以处理Channel 的I/O 以及数据的ChannelInboundHandler* .handler(**new** SimpleChannelInboundHandler<DatagramPacket>() {  @Override  **public void** channelRead0(ChannelHandlerContext ctx,  DatagramPacket msg) **throws** Exception {  *// Do something with the packet* }  });   *// 调用bind()方法，因为该协议是无连接的* ChannelFuture future = bootstrap.bind(**new** InetSocketAddress(**0**));  future.addListener(**new** ChannelFutureListener() {  @Override  **public void** operationComplete(ChannelFuture channelFuture) **throws** Exception {  **if** (channelFuture.isSuccess()) {  System.***out***.println(**"Channel bound"**);  } **else** {  System.***err***.println(**"Bind attempt failed"**);  channelFuture.cause().printStackTrace();  }  }  }); } |

## 8.8 关闭

引导使你的应用程序启动并且运行起来，但是迟早你都需要优雅地将它关闭。当然，你也可以让JVM 在退出时处理好一切，但是这不符合优雅的定义，优雅是指干净地释放资源。关闭Netty应用程序并没有太多的魔法，但是还是有些事情需要记在心上。

最重要的是，你需要关闭EventLoopGroup，它将处理任何挂起的事件和任务，并且随后释放所有活动的线程。这就是调用EventLoopGroup.shutdownGracefully()方法的作用。这个方法调用将会返回一个Future，这个Future 将在关闭完成时接收到通知。需要注意的是，shutdownGracefully()方法也是一个异步的操作，所以你需要阻塞等待直到它完成，或者向所返回的Future 注册一个监听器以在关闭完成时获得通知。

代码清单8-9 符合优雅关闭的定义。

|  |
| --- |
| **// 代码清单8-9 优雅关闭**  **public void** bootstrap() {  *// 创建处理I/O 的EventLoopGroup* EventLoopGroup group = **new** NioEventLoopGroup();  *// 创建一个Bootstrap类的实例并配置它* Bootstrap bootstrap = **new** Bootstrap();  bootstrap.group(group)  .channel(NioSocketChannel.**class**)  *//...* .handler(**new** SimpleChannelInboundHandler<ByteBuf>() {  @Override  **protected void** channelRead0(  ChannelHandlerContext channelHandlerContext,  ByteBuf byteBuf) **throws** Exception {  System.***out***.println(**"Received data"**);  }  });  bootstrap.connect(**new** InetSocketAddress(**"www.manning.com"**, **80**)).syncUninterruptibly();  *//,,,  // shutdownGracefully()方法将释放所有的资源，并且关闭所有的当前正在使用中的Channel* Future<?> future = group.shutdownGracefully();  *// block until the group has shutdown* future.syncUninterruptibly(); } |

或者，你也可以在调用EventLoopGroup.shutdownGracefully()方法之前，显式地在所有活动的Channel 上调用Channel.close()方法。但是在任何情况下，都请记得关闭EventLoopGroup 本身。