# Channel接口

Channel 是 Java NIO 的一个基本构造，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。

基本的 I/O 操作（bind()、 connect()、 read()和 write()）依赖于底层网络传输所提  
供的原语。在基于 Java 的网络编程中，其基本的构造是 class Socket。 Netty 的 Channel 接  
口所提供的 API，大大地降低了直接使用 Socket 类的复杂性。此外， Channel 也是拥有许多  
预定义的、专门化实现的广泛类层次结构的根，下面是一个简短的部分清单：  
ν EmbeddedChannel；  
ν LocalServerChannel；  
ν NioDatagramChannel；  
ν NioSctpChannel；  
ν NioSocketChannel。

Netty 的 Channel 实现是线程安全的，因此你可以存储一个到 Channel 的引用，并且每当你需要向远程节点写数据时， 都可以使用它， 即使当时许多线程都在使用它。

Channel被创建大概意思是 bind() 或connect()

bind()

|  |
| --- |
| 绑定 Channel 并返回一个 ChannelFuture，其将会在绑 定操作完成后接收到通知，在那之后必须调用 Channel. connect()方法来建立连接 |

connect()

|  |
| --- |
| 连接到远程节点并返回一个 ChannelFuture，其将 会在 连接操作完成后接收到通知 |

# 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。

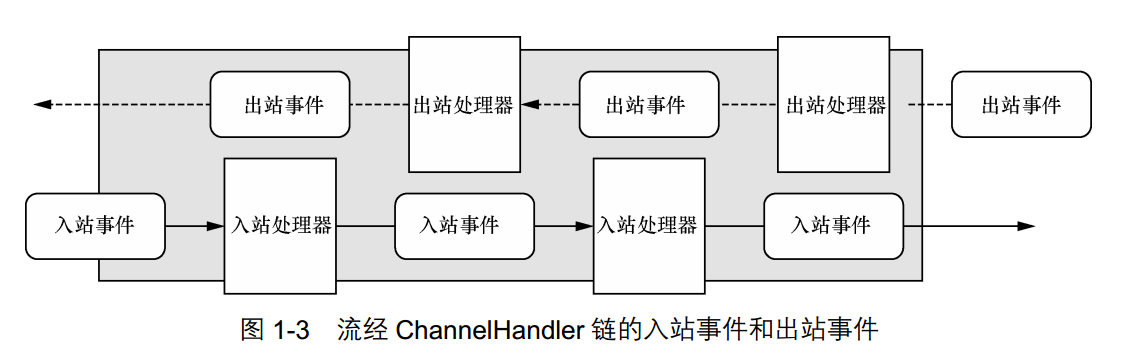
# Future ChannelFuture 接口

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。

Netty 提供了ChannelFuture 接口，其 addListener()方法注册了一个 ChannelFutureListener，以  
便在某个操作完成时（无论是否成功）得到通知。

# 事件和 ChannelHandler

每个事件都可以被分发给 ChannelHandler 类中的某个用户实现的方法。



ChannelHandler，是一个借口族的父借口，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。

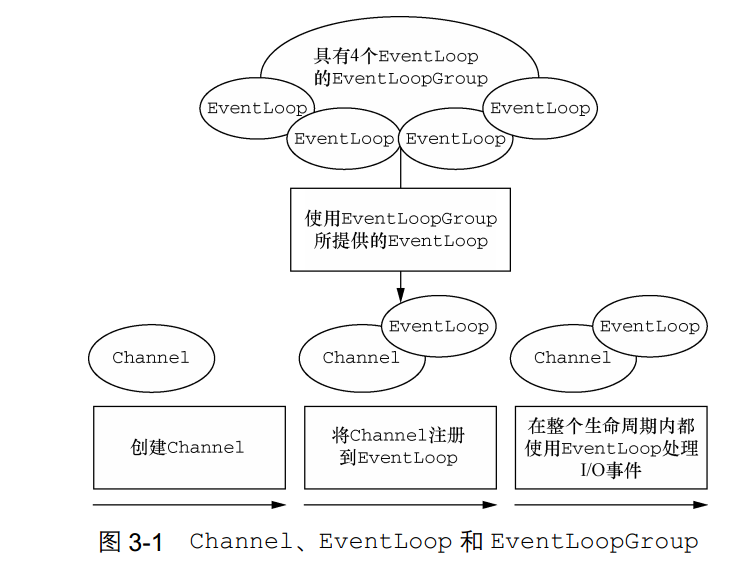
在Echo服务器中，会响应传入的消息，所以它需要实现ChannelInboundHandler接口，用来定义响应入站事件的方法。

# Channel、 EventLoop 和 ChannelFuture

Channel—Socket；  
EventLoop—控制流、多线程处理、并发；  
ChannelFuture—异步通知。

# EventLoop 接口

EventLoop 定义了 Netty 的核心抽象，用于处理连接的生命周期中所发生的事件。



这些关系是：   
ν 一个 EventLoopGroup 包含一个或者多个 EventLoop；  
ν 一个 EventLoop 在它的生命周期内只和一个 Thread 绑定；  
ν 所有由 EventLoop 处理的 I/O 事件都将在它专有的 Thread 上被处理；  
ν 一个 Channel 在它的生命周期内只注册于一个 EventLoop；  
ν 一个 EventLoop 可能会被分配给一个或多个 Channel。

单个EventLoop可能会被指派用于服务多个Channel。  
注意，在这种设计中，一个给定 Channel 的 I/O 操作都是由相同的 Thread 执行的，实际  
上消除了对于同步的需要。

# Netty4中的IO和事件处理

在Netty 4 中， 所有的I/O操作和事件都由已经被分配给了EventLoop的那个Thread来处理。

# Netty 3 中的 I/O 操作

只保证了入站（之前称为上游）事件会在所谓的 I/O 线程（对应于 Netty 4 中的 EventLoop）中执行。所有的出站（下游）事件都由调用线程处理，其可能是 I/O 线程也可能是别的线程。

# 任务调度

## 使用 EventLoop 调度任务

ScheduledExecutorService 的实现具有局限性，例如，事实上作为线程池管理的一部分，将会有额外的线程创建。如果有大量任务被紧凑地调度， 那么这将成为一个瓶颈。 Netty 通过 Channel 的 EventLoop 实现任务调度解决了这一问题。

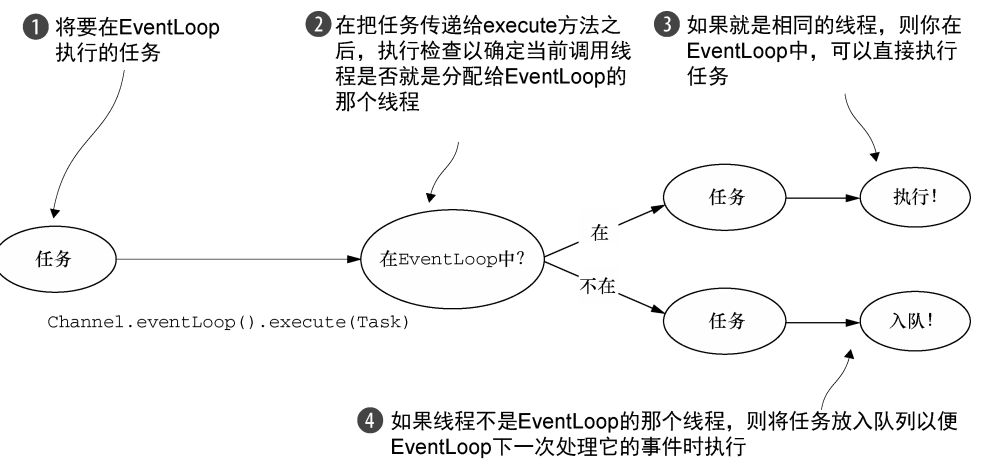
永远不要在 Netty 的 I/O 线程上执行任何非 CPU 限定  
的代码——你将会从 Netty 偷取宝贵的资源，并因此影响到服务器的吞吐量。  
因此， HttpRequest 和 HttpChunk 都可以通过切换到另一个不同的线程，来将执行流程  
移交给请求处理器。当请求处理器不是 CPU 限定时，就会发生这样的情况，不管是因为它们访  
问了数据库，还是执行了不适合于本地内存或者 CPU 的逻辑。

## 线程管理

Netty线程模型的卓越性能取决于对于当前执行的Thread的身份的确定，也就是说，确定它是否是分配给当前Channel以及它的EventLoop的那一个线程。

如果（当前）调用线程正是支撑 EventLoop 的线程，那么所提交的代码块将会被（直接）执行。否则，EventLoop 将调度该任务以便稍后执行，并将它放入到内部队列中。当 EventLoop下次处理它的事件时，它会执行队列中的那些任务/事件。这也就解释了任何的 Thread 是如何与 Channel 直接交互而无需在 ChannelHandler 中进行额外同步的。

注意，每个 EventLoop 都有它自已的任务队列，独立于任何其他的 EventLoop。



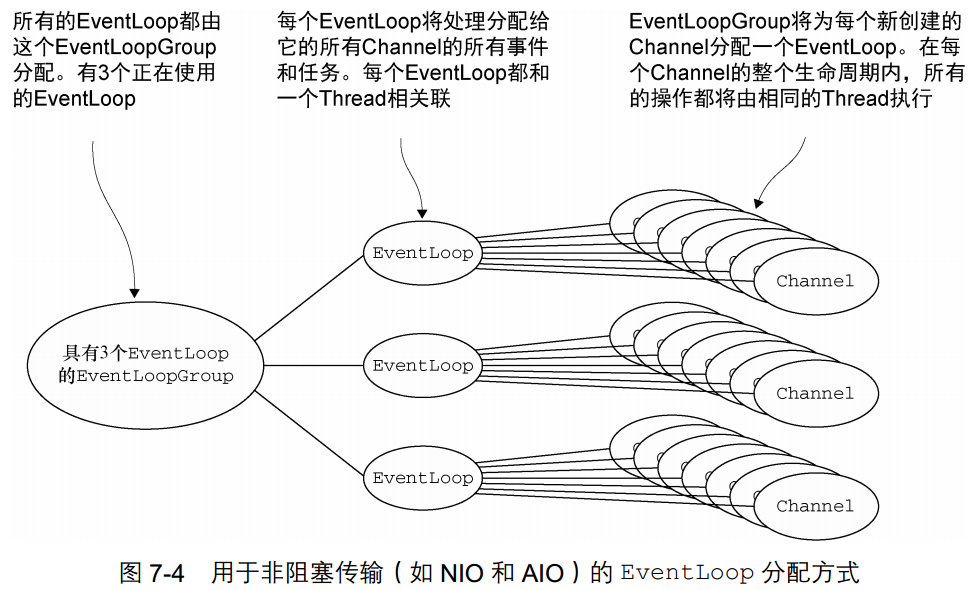
我们之前已经阐明了不要阻塞当前 I/O 线程的重要性。我们再以另一种方式重申一次：“永远不要将一个长时间运行的任务放入到执行队列中，因为它将阻塞需要在同一线程上执行的任何其他任务。” 如果必须要进行阻塞调用或者执行长时间运行的任务，我们建议使用一个专门的EventExecutor。

## EventLoop/线程的分配

服务于Channel的I/O和事件的EventLoop包含在EventLoopGroup中。

1.异步传输

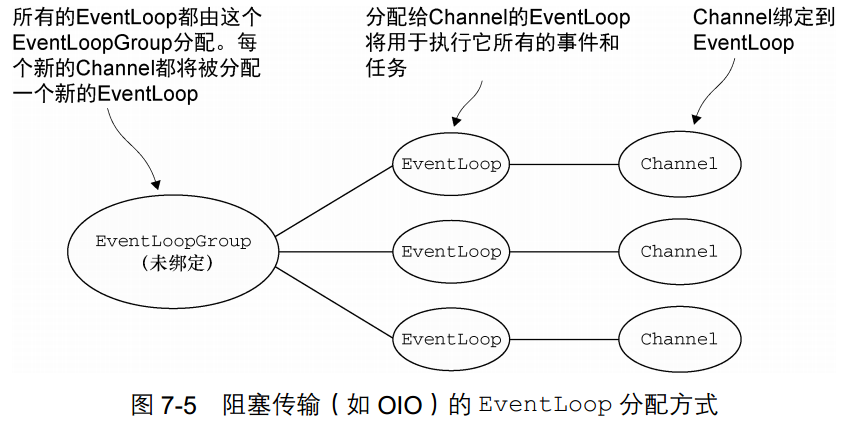
使用少量的EventLoop,每个EventLoop可以被多个Channel所共享。



EventLoopGroup负责为每个新创建的Channel分配一个EventLoop。在当前实现中，使用顺序循环(round-robin)的方式进行分配以获取一个均衡的分布，并且相同的EventLoop可能会分配给多个Channel。

2.阻塞传输

用于OIO，每一个Channel都将分配一个EventLoop(以及该EventLopp的线程)。



# Netty I/O线程

Netty的I/O线程NioEventLoop由于聚合了多路复用器Selector，可以同时并发处理成百上千个客户端SocketChannel。由于读写操作都是非阻塞的，这就可以充分提升I/O线程的运行效率，避免由频繁的I/O阻塞导致的线程挂起。另外，由于Netty采用了异步通信模式，一个I/O线程可以并发处理N个客户端连接和读写操作。

# ChannelHandler 接口

Netty 的主要组件是 ChannelHandler， 它充当了所有处理入站和出站数据的应用程序逻辑的容器。

Netty 以适配器类的形式提供了大量默认的 ChannelHandler 实现为什么需要适配器类  
有一些适配器类可以将编写自定义的 ChannelHandler 所需要的努力降到最低限度，因为它们提  
供了定义在对应接口中的所有方法的默认实现。  
下面这些是编写自定义 ChannelHandler 时经常会用到的适配器类：  
ν ChannelHandlerAdapter  
ν ChannelInboundHandlerAdapter  
ν ChannelOutboundHandlerAdapter  
ν ChannelDuplexHandler

ChannelPipeline 持有所有将应用于入站和出站数据以及事件的 ChannelHandler 实例，这些 ChannelHandler 实现了应用程序用于处理状态变化以及数据处理的逻辑。  
ChannelHandler 的典型用途包括：  
ν 将数据从一种格式转换为另一种格式；  
ν 提供异常的通知；  
ν 提供 Channel 变为活动的或者非活动的通知；  
ν 提供当 Channel 注册到 EventLoop 或者从 EventLoop 注销时的通知；  
ν 提供有关用户自定义事件的通知。

# ChannelPipeline 接口

ChannelPipeline 提供了 ChannelHandler 链的容器，并定义了用于在该链上传播入站和出站事件流的 API。

当 Channel 被创建时，它会被自动地分配到它专属的 ChannelPipeline。ChannelHandler 安装到 ChannelPipeline 中的过程如下所示：  
ν 一个ChannelInitializer的实现被注册到了ServerBootstrap中；  
ν 当 ChannelInitializer.initChannel()方法被调用时， ChannelInitializer将在 ChannelPipeline 中安装一组自定义的 ChannelHandler；  
ν ChannelInitializer 将它自己从 ChannelPipeline 中移除。

ChannelPipeline是一个拦截流经Channel的入站和出站事件的ChannelHandler 实例链.

每一个新创建的 Channel 都将会被分配一个新的 ChannelPipeline。这项关联是永久性的； Channel 既不能附加另外一个 ChannelPipeline，也不能分离其当前的。在 Netty 组件的生命周期中，这是一项固定的操作，不需要开发人员的任何干预。

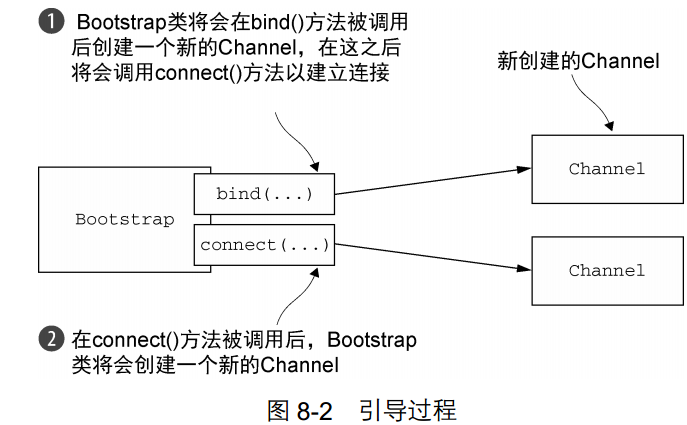
根据事件的起源，事件将会被 ChannelInboundHandler 或者 ChannelOutboundHandler处理。随后， 通过调用 ChannelHandlerContext 实现，它将被转发给同一超类型的下一个ChannelHandler.

# 引导类Bootstrap

引导类实现Cloneable，因为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，是一个浅拷贝。

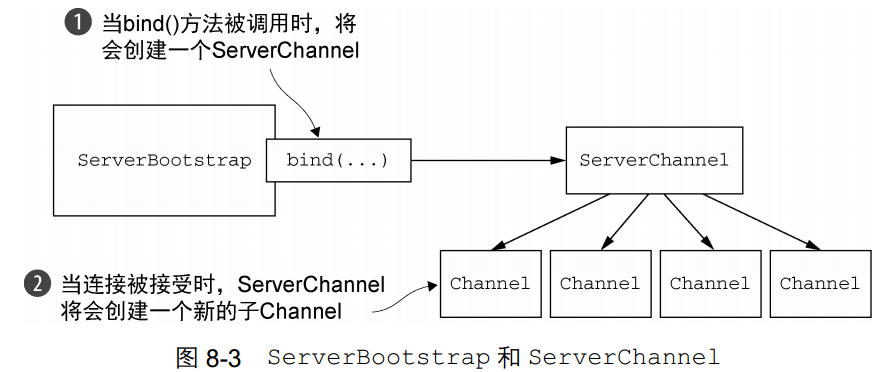
## 引导客户端

Bootstrap 类负责为客户端和使用无连接协议的应用程序创建 Channel



## 引导服务器

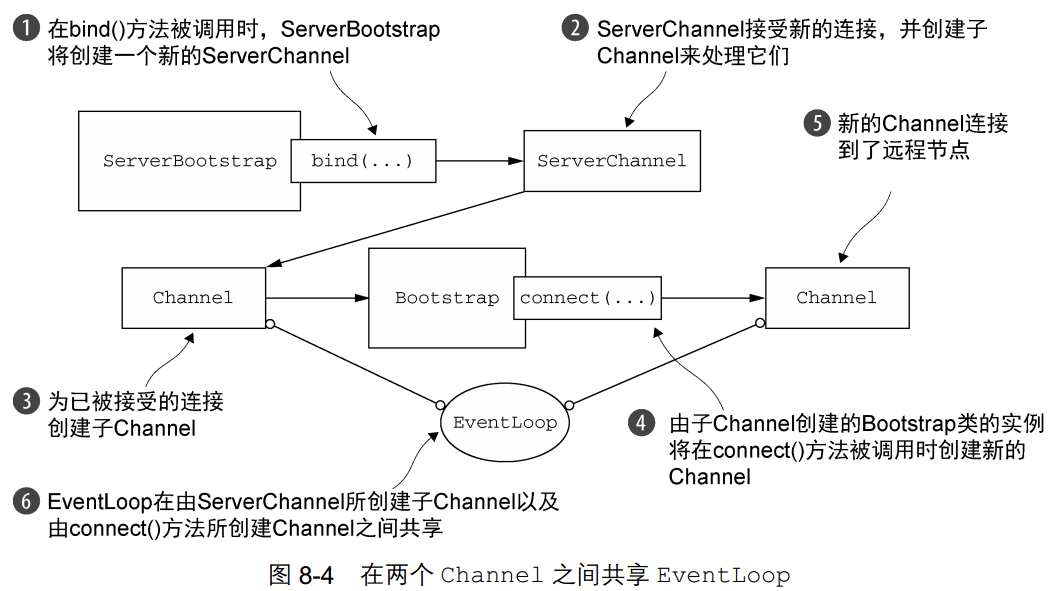
ServerChannel的实现负责创建子Channel，这些字Channel代表了已被接受的连接。因此，负责引导ServerChannel的ServerBootstrap提供了这些方法，以简化将设置应用到已被接受的子Channel的ChannelConfig的任务。



## 从Channel引导客户端

现在服务端收到一个请求，要求服务开启另一个客户端，传统的方法是从接受到客户端的请求的Channel中引导一个客户端Channel,然后你创建一个Bootstrap实例，一个EventLoop但是不高效，因为它将要求你为每一个新创建的客户端Channel定义另一个EventLoop,产生额外的线程，以及接受到客户端的请求的Channel和新开启的Channel之间交换数据时不可避免的上下文切换。

好的方案，将从接受到客户端的请求的Channel的EventLoop传递给新的Bootstap使用，节省线程。



## 在引导过程中添加多个 ChannelHandler

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。

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。

# ChannelHandlerContext接口

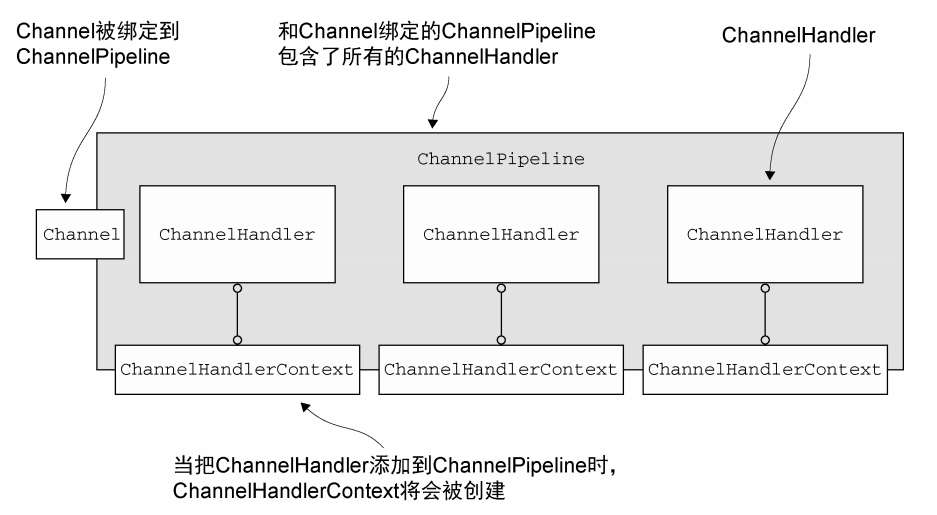
ChannelHandlerContext 代表了 ChannelHandler 和 ChannelPipeline 之间的关联，每当有 ChannelHandler 添加到 ChannelPipeline 中时，都会创建 ChannelHandlerContext。 ChannelHandlerContext 的主要功能是管理它所关联的 ChannelHandler 和在同一个 ChannelPipeline 中的其他 ChannelHandler 之间的交互。

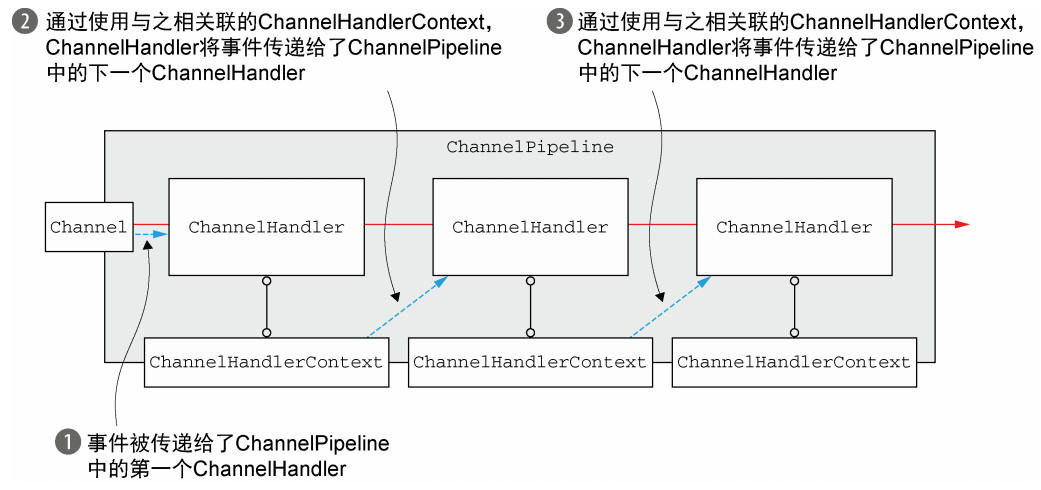
ChannelHandlerContext使得ChannelHandler能够和它的ChannelPipeline以及其他的ChannelHandler 交 互 。 ChannelHandler 可 以 通 知 其 所 属 的 ChannelPipeline 中 的 下 一 个ChannelHandler，甚至可以动态修改它所属的ChannelPipeline。

ChannelHandlerContext 有很多的方法，其中一些方法也存在于 Channel 和 ChannelPipeline 本身上，但是有一点重要的不同。如果调用 Channel 或者 ChannelPipeline 上的这些方法，它们将沿着整个 ChannelPipeline 进行传播。而调用位于 ChannelHandlerContext上的相同方法，则将从当前所关联的 ChannelHandler 开始，并且只会传播给位于该ChannelPipeline 中的下一个能够处理该事件的 ChannelHandler。

当使用 ChannelHandlerContext 的 API 的时候，请牢记以下两点：  
ν ChannelHandlerContext 和 ChannelHandler 之间的关联（绑定）是永远不会改  
变的，所以缓存对它的引用是安全的；

ν 如同我们在本节开头所解释的一样，相对于其他类的同名方法， ChannelHandler Context的方法将产生更短的事件流， 应该尽可能地利用这个特性来获得最大的性能。





# Netty的使用

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）。

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ID存储为Channel的一个属性来完成。

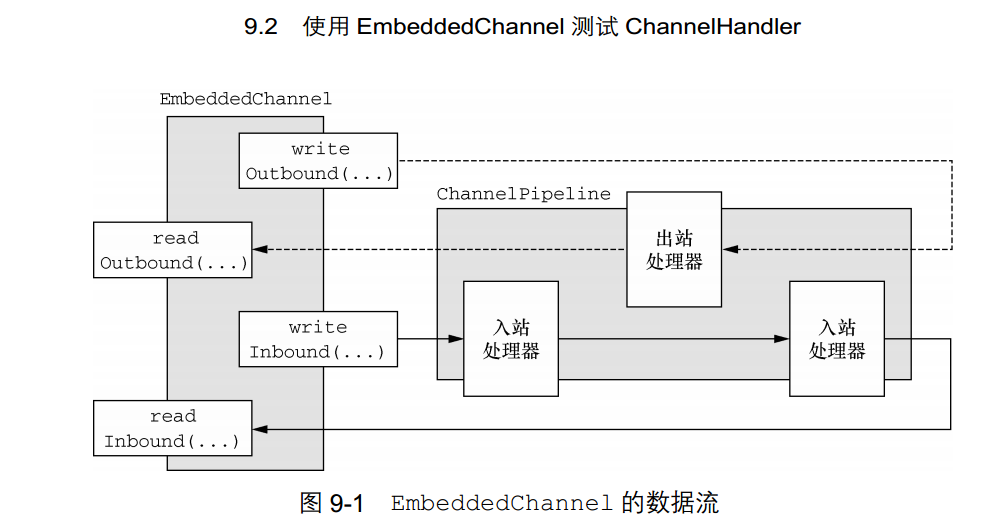
bootstrap.attr(id, 123456);

Integer idValue = channelHandlerContext.channel().attr(id).get();

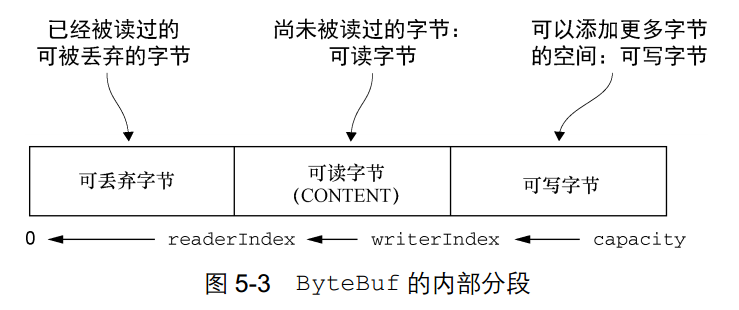
# 优雅关闭

Future<?> future = group.shutdownGracefully();  
// block until the group has shutdown  
future.syncUninterruptibly();

# 单元测试



# ByteBuf可丢弃字节



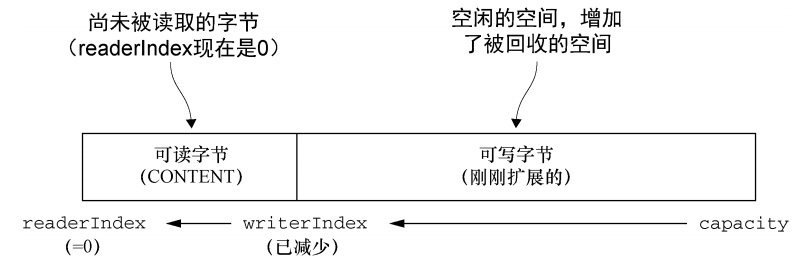


图 5-4 展示了图 5-3 中所展示的缓冲区上调用discardReadBytes()方法后的结果。可以看  
到， 可丢弃字节分段中的空间已经变为可写的了。注意，在调用discardReadBytes()之后，对  
可写分段的内容并没有任何的保证

虽然你可能会倾向于频繁地调用 discardReadBytes()方法以确保可写分段的最大化，但是  
请注意， 这将极有可能会导致内存复制，因为可读字节（图中标记为 CONTENT 的部分）必须被移动到缓冲区的开始位置。

不要频繁的使用discardReadBytes()，极有可能会导致内存复制。