- NETTY

精尽 Netty 源码分析 —— Netty 简介（二）之核心组件

1. 概述

什么是 Netty ？

> Netty 是一款提供异步的、事件驱动的网络应用程序框架和工具，用以快速开发高性能、高可靠性的网络服务器和客户端程序。
>
> 也就是说，Netty 是一个基于 NIO 的客户、服务器端编程框架。使用 Netty 可以 网络应用，例如实现了某种协议的客户，服务端 了网络应用的编程开发过程，例如，TCP 和 UDP

文章目录

设计	1. 统一的 API ， 支持多种传输类型(阻塞和非阻塞的) 2. 简单而强大的线程模型 3. 真正的无连接数据报套接字(UDP)支持 4. 连接逻辑组件(ChannelHander 中顺序处理消息)以及组件复用(一个 ChannelHandel 可以被多个ChannelPipeLine 复用)
易于使用	1. 详实的 Javadoc 和大量的示例集 2. 不需要超过 JDK 1.6+ 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)，更低的资源消耗以及最少的内存复制
健壮性	1. 不会因为慢速、快速或者超载的连接而导致 OutOfMemoryError 2. 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

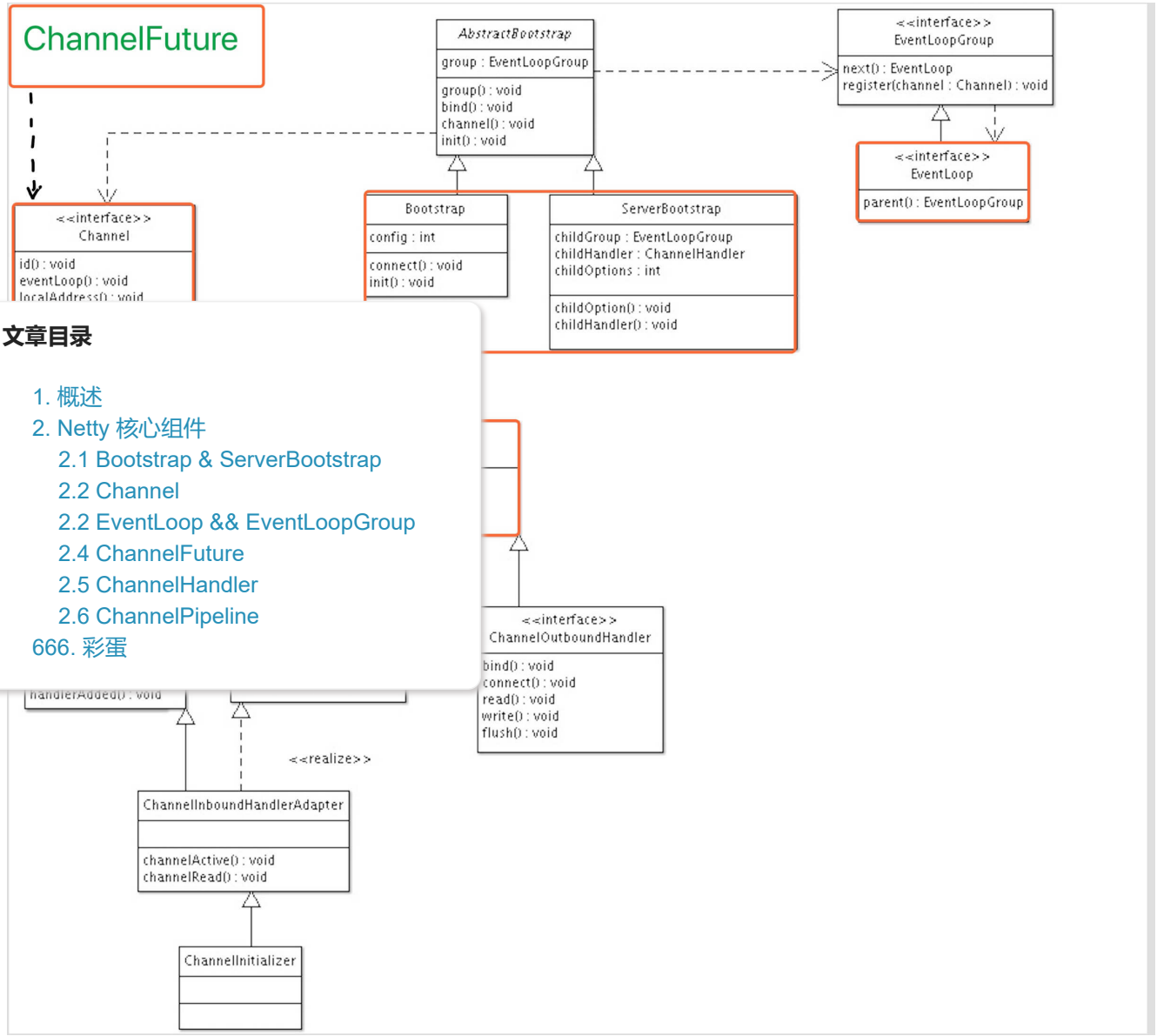
2. Netty 核心组件

为了后期更好地理解和进一步深入 Netty，有必要总体认识一下 Netty 所用到的核心组件以及他们在整个 Netty 架构中是如何协调工作的。

Netty 有如下几个核心组件：

- Bootstrap & ServerBootstrap
- Channel
- ChannelFuture
- EventLoop & EventLoopGroup
- ChannelHandler
- ChannelPipeline

核心组件的高层类图如下：



高层类图

2.1 Bootstrap & ServerBootstrap

这 2 个类都继承了AbstractBootstrap，因此它们有很多相同的方法和职责。它们都是**启动器**，能够帮助 Netty 使用者更加方便地组装和配置 Netty ， 也可以更方便地启动 Netty 应用程序。相比使用者自己从头去将 Netty 的各部分组装起来要方便得多，

降低了使用者的学习和使用成本。它们是我们使用 Netty 的入口和最重要的 API ，可以通过它来连接到一个主机和端口上，也可以通过它来绑定到一个本地的端口上。总的来说，**它们两者之间相同之处要大于不同。**

> 老艿艿：Bootstrap & ServerBootstrap 对于 Netty ，就相当于 Spring Boot 是 Spring 的启动器。

它们和其它组件之间的关系是它们将 Netty 的其它组件进行组装和配置，所以它们会组合和直接或间接依赖其它的类。

Bootstrap 用于启动一个 Netty TCP 客户端，或者 UDP 的一端。

- 通常使用 `#connet(...)` 方法连接到远程的主机和端口，作为一个 Netty TCP 客户端。
- 也可以通过 `#bind(...)` 方法绑定本地的一个端口，作为 UDP 的一端。
- 仅仅需要使用**一个** EventLoopGroup 。

ServerBootstrap 往往是用于启动一个 Netty 服务端。

- 通常使用 `#bind(...)` 方法绑定本地的端口上，然后等待客户端的连接。
- 使用**两个** EventLoopGroup 对象(当然这个对象可以引用同一个对象)：第一个用于处理它本地 Socket **连接**的 IO 事件处理，而第二个责负责处理远程客户端的 IO 事件处理。

2.2 Channel

的 I/O 操作，如 bind、connect、read、write 之外，还包括了 Netty 框架

程序员来说并不是那么友好，直接使用其成本还是稍微高了点。而 Netty 直接与 Socket 进行操作的复杂性。而相对于原生 NIO 的 Channel，Netty 二版)》)：

封装，将网络 I/O 操作、网络 I/O 相关联的其他操作封装起来，统一对

nel 和 ServerSocketChannel 提供统一的视图，由不同子类实现不同的实现功能和接口的重用。

- 具体实现采用聚合而非包含的方式，将相关的功能类聚合在 Channel 中，由 Channel 统一负责和调度，功能实现更加灵活。

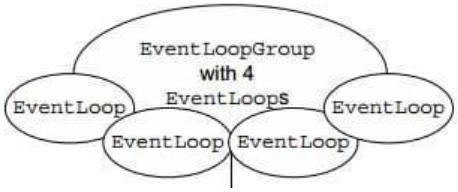
2.2 EventLoop && EventLoopGroup

Netty 基于**事件驱动模型**，使用不同的事件来通知我们状态的改变或者操作状态的改变。它定义了在整个连接的生命周期里当有事件发生的时候处理的核心抽象。

Channel 为Netty 网络操作抽象类，EventLoop 负责处理注册到其上的 Channel 处理 I/O 操作，两者配合参与 I/O 操作。

EventLoopGroup 是一个 EventLoop 的分组，它可以获取到一个或者多个 EventLoop 对象，因此它提供了迭代出 EventLoop 对象的方法。

下图是 Channel、EventLoop、Thread、EventLoopGroup 之间的关系(摘自《Netty In Action》)：



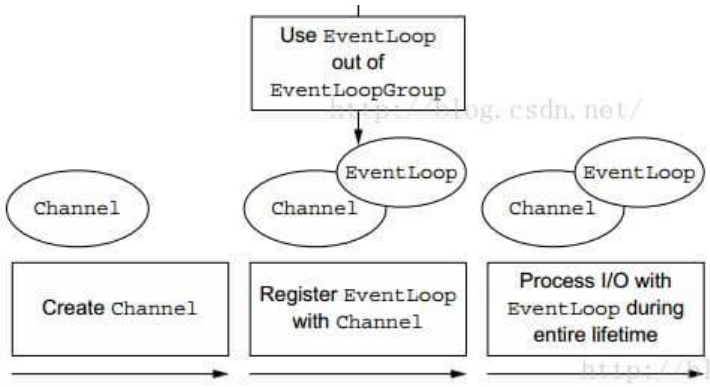


Figure 3.1 Channels, EventLoops, and EventLoopGroups

Channel、EventLoop、Thread、EventLoopGroup

- 一个 EventLoopGroup 包含一个或多个 EventLoop ，即 EventLoopGroup : EventLoop = `1 : n` 。
- 一个 EventLoop 在它的生命周期内，只能与一个 Thread 绑定，即 EventLoop : Thread = `1 : 1` 。
- 所有有 EventLoop 处理的 I/O 事件都将在它**专有**的 Thread 上被处理，从而保证线程安全，即 Thread : EventLoop = `1 : 1` 。
- 一个 Channel 在它的生命周期内只能注册到一个 EventLoop 上，即 Channel : EventLoop = `n : 1` 。
- 一个 EventLoop 可被分配至一个或多个 Channel ，即 EventLoop : Channel = `1 : n` 。

当一个连接到达时，Netty 就会创建一个 Channel，然后从 EventLoopGroup 中分配一个 EventLoop 来给这个 Channel 绑定上，在该 Channel 的整个生命周期中都是有这个绑定的 EventLoop 来服务的。

2.4 ChannelFuture

文章目录

因此，我们不能立刻得知消息是否已经被处理了。Netty 提供了 `r(...)` 方法，注册一个 ChannelFutureListener，当操作执行成功或者

y 中**最常用**的组件。ChannelHandler 主要用来处理各种事件，这里的事件换等。

andler 和 ChannelOutboundHandler，其中 ChannelInboundHandler 用 nnelOutboundHandler 则相反，用于接收、处理出站(Outbound)的数据

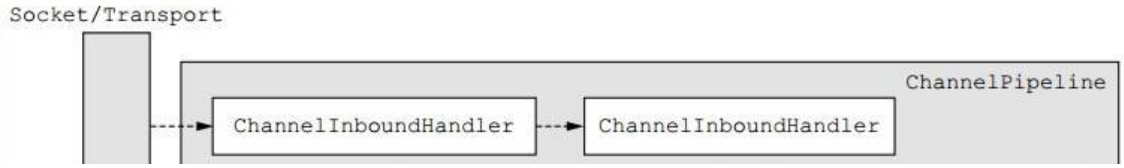
- ChannelInboundHandler 的实现类还包括一系列的 **Decoder** 类，对输入字节流进行解码。
- ChannelOutboundHandler 的实现类还包括一系列的 **Encoder** 类，对输入字节流进行编码。

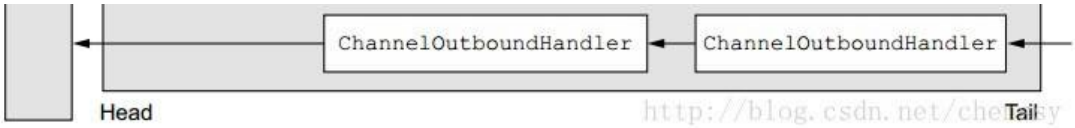
ChannelDuplexHandler 可以**同时**用于接收、处理入站和出站的数据和时间。

ChannelHandler 还有其它的一系列的抽象实现 Adapter ，以及一些用于编解码具体协议的 ChannelHandler 实现类。

2.6 ChannelPipeline

ChannelPipeline 为 ChannelHandler 的**链**，提供了一个容器并定义了用于沿着链传播入站和出站事件流的 API 。一个数据或者事件可能会被多个 Handler 处理，在这个过程中，数据或者事件经流 ChannelPipeline ，由 ChannelHandler 处理。在这个处理过程中，一个 ChannelHandler 接收数据后处理完成后交给下一个 ChannelHandler，或者什么都不做直接交给下一个 ChannelHandler。





ChannelPipeline

- 当一个数据流进入 ChannelPipeline 时，它会从 ChannelPipeline 头部开始，传给第一个 ChannelInboundHandler 。当第一个处理完后再传给下一个，一直传递到管道的尾部。
- 与之相对应的是，当数据被写出时，它会从管道的尾部开始，先经过管道尾部的“最后”一个ChannelOutboundHandler，当它处理完成后会传递给前一个 ChannelOutboundHandler 。

上图更详细的，可以是如下过程：

```
*                                      I/O Request
*                                   via {@link Channel} or
*                                {@link ChannelHandlerContext}
*                                           |
*  +-----------------------------------------+---------------+
*  |                          ChannelPipeline |               |
*  |                                         \|/              |
*  |    +---------------------+           +-----------+----------+    |
*  |    | Inbound Handler  N  |           | Outbound Handler  1  |    |
*  |    +----------+----------+           +-----------+----------+    |
*  |              /|\                                |               |
*  |               |                                \|/              |
*  |    +----------+----------+           +-----------+----------+    |
*  |    | Inbound Handler N-1 |           | Outbound Handler  2  |    |
*  |    +----------+----------+           +-----------+----------+    |
*  |              /|\                                .               |
*  |               .                                 .               |
*  | ChannelHandlerContext.fireIN_EVT() ChannelHandlerContext.OUT_EVT()|
*  |        [ method call]                       [method call]       |
*  |               .                                 .               |
*  |               .                                \|/              |
*  |    +----------+----------+           +-----------+----------+    |
*  |    | Inbound Handler  2  |           | Outbound Handler M-1 |    |
*  |    +----------+----------+           +-----------+----------+    |
*  |              /|\                                |               |
*  |               |                                \|/              |
*  |    +----------+----------+           +-----------+----------+    |
*  |    | Inbound Handler  1  |           | Outbound Handler  M  |    |
*  |    +----------+----------+           +-----------+----------+    |
*  |              /|\                                |               |
*  +---------------+-----------------------------------+---------------+
*                  |                                \|/
*  +---------------+-----------------------------------+---------------+
*  |               |                                   |               |
*  |       [ Socket.read() ]                    [ Socket.write() ]     |
*  |                                                                   |
*  |  Netty Internal I/O Threads (Transport Implementation)            |
*  +-------------------------------------------------------------------+
```

文章目录

当 ChannelHandler 被添加到 ChannelPipeline 时，它将会被分配一个 **ChannelHandlerContext** ，它代表了 ChannelHandler 和 ChannelPipeline 之间的绑定。其中 ChannelHandler 添加到 ChannelPipeline 中，通过 ChannelInitializer 来实现，过程如

下：

1. 一个 ChannelInitializer 的实现对象，被设置到了 BootStrap 或 ServerBootStrap 中。
2. 当 `ChannelInitializer#initChannel()` 方法被调用时，ChannelInitializer 将在 ChannelPipeline 中创建**一组**自定义的 ChannelHandler 对象。
3. ChannelInitializer 将它自己从 ChannelPipeline 中移除。

> ChannelInitializer 是一个特殊的 ChannelInboundHandlerAdapter 抽象类。

666. 彩蛋

本文整理于如下两篇文章：

- 小明哥 《【死磕 Netty 】—— Netty的核心组件》
- 杨武兵 《Netty 源码分析系列 —— 概述》
- 乒乓狂魔 《Netty 源码分析（一）概览》

😈 见谅，不擅长写理论型的内容哈。

文章目录