

Netty4.x 中文教程系列

Netty 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。

Netty 是一个 NIO 客户端 服务端框架。允许快速简单的开发网络应用程序。例如：服务端和客户端之间的协议。它最棒的地方在于简化了网络编程规范。例如:TCP 和 UDP socket 服务。

2.创建 Server 服务端

Netty 创建全部都是实现自 [AbstractBootstrap](#)。客户端的是 Bootstrap，服务端的则是 ServerBootstrap。

2.1 创建一个 HelloServer

```
package org.example.hello;

import io.netty.bootstrap.ServerBootstrap;
import io.netty.channel.ChannelFuture;
import io.netty.channel.EventLoopGroup;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.nio.NioServerSocketChannel;

public class HelloServer {

    /**
     * 服务端监听的端口地址
     */
    private static final int portNumber = 7878;

    public static void main(String[] args) throws InterruptedException {
        EventLoopGroup bossGroup = new NioEventLoopGroup();
        EventLoopGroup workerGroup = new NioEventLoopGroup();
        try {
            ServerBootstrap b = new ServerBootstrap();
            b.group(bossGroup, workerGroup);
            b.channel(NioServerSocketChannel.class);
            b.childHandler(new HelloServerInitializer());

            // 服务器绑定端口监听
            ChannelFuture f = b.bind(portNumber).sync();
            // 监听服务器关闭监听
            f.channel().closeFuture().sync();

            // 可以简写为
            /* b.bind(portNumber).sync().channel().closeFuture().sync(); */
        } finally {
            bossGroup.shutdownGracefully();
        }
    }
}
```

```

        workerGroup.shutdownGracefully();
    }
}
}

```

EventLoopGroup 是在 4.x 版本中提出来的一个新概念。用于 channel 的管理。服务端需要两个。和 3.x 版本一样，一个是 boss 线程一个是 worker 线程。

```
b.childHandler(new HelloServerInitializer());
```

服务端简单的代码，真的没有办法在精简了感觉。就是一个绑定端口操作。

2.2 创建和实现 HelloServerInitializer

在 HelloServer 中的 HelloServerInitializer 在这里实现。

首先我们需要明确我们到底是要做什么的。很简单。HelloWorld!。我们希望实现一个能够像服务端发送文字的功能。服务端假如可以最好还能返回点消息给客户端，然客户端去显示。

需求简单。那我们下面就准备开始实现。

DelimiterBasedFrameDecoder Netty 在官方网站上提供的示例显示 有这么一个解码器可以简单的消息分割。

其次 在 decoder 里面我们找到了 String 解码编码器。着都是官网提供给我们的。

```
package org.example.hello;
```

```
import io.netty.channel.ChannelInitializer;
import io.netty.channel.ChannelPipeline;
import io.netty.channel.socket.SocketChannel;
import io.netty.handler.codec.DelimiterBasedFrameDecoder;
import io.netty.handler.codec.Delimiters;
import io.netty.handler.codec.string.StringDecoder;
import io.netty.handler.codec.string.StringEncoder;
```

```
public class HelloServerInitializer extends ChannelInitializer<SocketChannel> {
```

```
    @Override
```

```
    protected void initChannel(SocketChannel ch) throws Exception {
        ChannelPipeline pipeline = ch.pipeline();
```

```
        // 以("\n")为结尾分割的 解码器
```

```
        pipeline.addLast("framer", new DelimiterBasedFrameDecoder(8192,
Delimiters.lineDelimiter()));
```

```
        // 字符串解码 和 编码
```

```
        pipeline.addLast("decoder", new StringDecoder());
        pipeline.addLast("encoder", new StringEncoder());
```

```
        // 自己的逻辑 Handler
```

```
        pipeline.addLast("handler", new HelloServerHandler());
```

```
    }
}
```

上面的三个解码和编码都是系统。

另外我们自己的 Handler 怎么办呢。在最后我们添加一个自己的 Handler 用于写自己的处理逻辑。

2.3 增加自己的逻辑 HelloServerHandler

自己的 Handler 我们这里先去继承 extends 官网推荐的 SimpleChannelInboundHandler<C>。在这里 C，由于我们需求里面发送的是字符串。这里的 C 改写为 String。

```
package org.example.hello;

import java.net.InetAddress;

import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.SimpleChannelInboundHandler;

public class HelloServerHandler extends SimpleChannelInboundHandler<String> {

    @Override
    protected void channelRead0(ChannelHandlerContext ctx, String msg) throws Exception {
        // 收到消息直接打印输出
        System.out.println(ctx.channel().remoteAddress() + " Say : " + msg);

        // 返回客户端消息 - 我已经接收到了你的消息
        ctx.writeAndFlush("Received your message !\n");
    }

    /**
     *
     * 覆盖 channelActive 方法 在 channel 被启用的时候触发 (在建立连接的时候)
     *
     * channelActive 和 channelInactive 在后面的内容中讲述，这里先不做详细的描述
     */
    @Override
    public void channelActive(ChannelHandlerContext ctx) throws Exception {

        System.out.println("RemoteAddress : " + ctx.channel().remoteAddress() + " active !");

        ctx.writeAndFlush( "Welcome to " + InetAddress.getLocalHost().getHostName() + "
service!\n");

        super.channelActive(ctx);
    }
}
```

在 channelHandlerContext 自带一个 writeAndFlush 方法。方法的作用是写入 Buffer 并刷入。

注意:在 3.x 版本中此处有很大区别。在 3.x 版本中 write()方法是自动 flush 的。在 4.x 版本的前面几个版本也是一样的。但是在 4.0.9 之后修改为 WriteAndFlush。普通的 write 方法将不会发送消息。需要手动在 write 之后 flush()一次

这里 channelActive 的意思是当连接活跃(建立)的时候触发.输出消息源的远程地址。并返回欢迎消息。

channelRead0 在这里的作用是类似于 3.x 版本的 messageReceived()。可以当做是每一次收到消息是触发。

我们在这里的代码是返回客户端一个字符串"Received your message !"。

注意:字符串最后面的"\n"是必须的。因为我们在前面的解码器 DelimiterBasedFrameDecoder 是一个根据字符串结尾为"\n"来结尾的。假如没有这个字符的话。解码会出现问题。

2.Client 客户端

类似于服务端的代码。我们不做特别详细的解释。

直接上示例代码：

```
package org.example.hello;

import io.netty.bootstrap.Bootstrap;
import io.netty.channel.Channel;
import io.netty.channel.EventLoopGroup;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.nio.NioSocketChannel;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class HelloClient {

    public static String host = "127.0.0.1";
    public static int port = 7878;

    /**
     * @param args
     * @throws InterruptedException
     * @throws IOException
     */
    public static void main(String[] args) throws InterruptedException, IOException {
        EventLoopGroup group = new NioEventLoopGroup();
        try {
            Bootstrap b = new Bootstrap();
            b.group(group)
              .channel(NioSocketChannel.class)
              .handler(new HelloClientInitializer());

            // 连接服务端
            Channel ch = b.connect(host, port).sync().channel();

            // 控制台输入
            BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
            for (;;) {
                String line = in.readLine();
                if (line == null) {
                    continue;
                }
            }
            /**
             * 向服务端发送在控制台输入的文本 并用"\r\n"结尾
             * 之所以用\r\n 结尾 是因为我们在 handler 中添加了 DelimiterBasedFrameDecoder 帧解码。
             */
        }
    }
}
```

* 这个解码器是一个根据\n 符号位分隔符的解码器。所以每条消息的最后必须加上\n 否则无法识别和解码

```
    **/  
    ch.writeAndFlush(line + "\r\n");  
    }  
    } finally {  
        // The connection is closed automatically on shutdown.  
        group.shutdownGracefully();  
    }  
}
```

下面的是 HelloClientInitializer 代码貌似是和服务端的完全一样。我没注意看。其实编码和解码是相对的。多以服务端和客户端都是解码和编码。才能通信。

```
package org.example.hello;
```

```
import io.netty.channel.ChannelInitializer;  
import io.netty.channel.ChannelPipeline;  
import io.netty.channel.socket.SocketChannel;  
import io.netty.handler.codec.DelimiterBasedFrameDecoder;  
import io.netty.handler.codec.Delimiters;  
import io.netty.handler.codec.string.StringDecoder;  
import io.netty.handler.codec.string.StringEncoder;
```

```
public class HelloClientInitializer extends ChannelInitializer<SocketChannel> {
```

```
    @Override
```

```
    protected void initChannel(SocketChannel ch) throws Exception {  
        ChannelPipeline pipeline = ch.pipeline();
```

```
        /*
```

```
        * 这个地方的 必须和服务端对应上。否则无法正常解码和编码
```

```
        *
```

```
        * 解码和编码 我将会在下一张为大家详细的讲解。再次暂时不做详细的描述
```

```
        *
```

```
        **/
```

```
        pipeline.addLast("framer", new DelimiterBasedFrameDecoder(8192,  
Delimiters.lineDelimiter()));
```

```
        pipeline.addLast("decoder", new StringDecoder());
```

```
        pipeline.addLast("encoder", new StringEncoder());
```

```
        // 客户端的逻辑
```

```
        pipeline.addLast("handler", new HelloClientHandler());
```

```
    }  
}
```

```
HelloClientHandler :
```

```
package org.example.hello;
```

```
import io.netty.channel.ChannelHandlerContext;  
import io.netty.channel.SimpleChannelInboundHandler;
```

```
public class HelloClientHandler extends SimpleChannelInboundHandler<String> {
```

```
    @Override
```

```

protected void channelRead0(ChannelHandlerContext ctx, String msg) throws Exception {

    System.out.println("Server say : " + msg);
}

@Override
public void channelActive(ChannelHandlerContext ctx) throws Exception {
    System.out.println("Client active ");
    super.channelActive(ctx);
}

@Override
public void channelInactive(ChannelHandlerContext ctx) throws Exception {
    System.out.println("Client close ");
    super.channelInactive(ctx);
}
}

```

1. HelloServer 详解

HelloServer 首先定义了一个静态终态的变量---服务端绑定端口 7878。至于为什么是这个 7878 端口，纯粹是笔者个人喜好。大家可以按照自己的习惯选择端口。当然了。常用的几个端口(例如:80,8080,843(Flash 及 Silverlight 策略文件请求端口等等)，3306(Mysql 数据库占用端口))最好就不要占用了，避免一些奇怪的问题。

HelloServer 类里面的代码并不多。只有一个 main 函数，加上内部短短的几行代码。

Main 函数开始的位置定义了两个工作线程，一个命名为 WorkerGroup，另一个命名为 BossGroup。都是实例化 NioEventLoopGroup。这一点和 3.x 版本中基本思路是一致的。Worker 线程用于管理线程为 Boss 线程服务。

讲到这里需要解释一下 EventLoopGroup，它是 4.x 版本提出来的一个新概念。类似于 3.x 版本中的线程。用于管理 Channel 连接的。在 main 函数的结尾就用到了 EventLoopGroup 提供的便捷的方法，shutdownGraceFully()，翻译为中文就是优雅的全部关闭。感觉是不是很有意思。作者居然会如此可爱的命名了这样一个函数。查看相应的源代码。我们可以在 DefaultEventExecutorGroup 的父类 MultithreadEventExecutorGroup 中看到它的实现代码。关闭了全部 EventExecutor 数组 child 里面子元素。相比于 3.x 版本这是一个比较重大的改动。开发者可以很轻松的全部关闭，而不需要担心出现内存泄露。

在 try 里面实例化一个 ServerBootstrap b。设置 group。设置 channel 为 NioServerSocketChannel。

设置 childHandler，在这里使用实例化一个 HelloServerInitializer 类来实现，继承 ChannelInitializer<SocketChannel>。内部的代码我们可以在前文的注视中大致了解一下，主要作用是设置相关的字节解码编码器，和代码处理逻辑。Handler 是 Netty 包里面占很大一个比例。可见其作用和用途。Handler 涉及很多领域。HTTP，UDP，Socket，WebSocket 等等。详细的部分在本章的第三节解释。

设置好 Handler 绑定端口 7878，并调用函数 sync()，监听端口(等待客户端连接和发送消息)。并监听端口关闭(为了防止线程停止)。

最后 finally 我们要优雅的全部关闭服务端。^_^

2. HelloClient 详解

相比于服务端的代码。客户端要精简一些。

客户端仅仅只需要一个 worker 的 EventLoopGroup。其次是类似于 ServerBootstrap 的 HandlerInitializer。

唯一不同的可能就是客户端的 connect 方法。服务端的绑定并监听端口，客户端是连接指定的地址。Sync().channel()是为了返回这个连接服务端的 channel，并用于后面代码的调用。

BufferedReader 这个是为了用于控制台输入的。不做详细的解释了。大家都懂的。

当用户输入一行内容并回车之后。循环的读取每一行内容。然后使用 writeAndFlush 向服务端发送消息。

3. HandlerInitializer 详解

Handler 在 Netty 中是一个比较重要的概念。有着相当重要的作用。相比于 Netty 的底层。我们接触更多的应该是他的 Handler。在这里我将它剥离出来单独解释。

ServerHandlerInitializer 继承与 ChannelInitializer<SocketChannel> 需要我们实现一个 initChannel() 方法。我们定义的 handler 就是写在这里面。

在最开始的地方定义了一个 DelimiterBasedFrameDecoder。按直接翻译就是基于分隔符的帧解码器。再一次感觉框架的作者的命名，好直接好简单。详细的内容我们在后面的文章中为大家详细的解释。目前大家知道他是以分隔符为分割标准的解码器就好了。

也许有人会问分隔符是什么？我只能 !*_ :“纳尼！！”。分隔符其实就是“\n”我们在学习 C 语言的时候最常用的也许就是这个分隔符了吧。

下面的则是 StringDecoder 和 StringEncoder。字符串解码器和字符串编码器。

最后面则是我们自己的逻辑。服务/客户端逻辑是在消息解码之后处理的。然后服务/客户端返回相关消息则是需要对消息进行相对应的编码。最终才是以二进制数据流的形式发送给服务/客户端的。

Netty4.x 中文教程系列(四) ChannelHandler

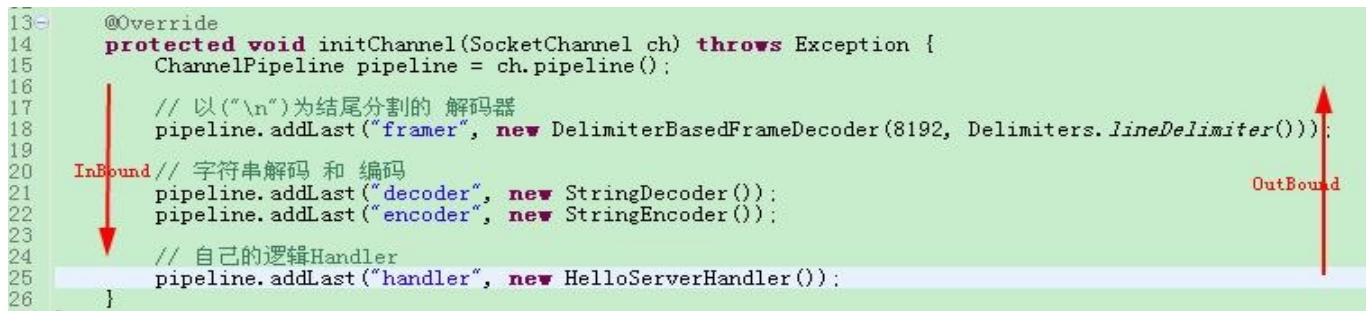
这篇文章用以解释 ChannelHandler。笔者本身在以前写过文章 [ChannelHandler 改动及影响](#) 和 [ChannelInitializer 学习](#) 对 Netty 的 ChannelHandler 做过阐述。里面主要描述了 4.x 版本相对于 3.x 版本的改动以及影响。并引用了一些文章。为大家详细的解释了 ChannelHandler 里面涉及架构。

1.在 4.x 版本中的 ChannelHandler

ChannelHandler 接口是 Handler 里面的最高的接口。

ChannelInboundHandler 接口和 ChannelOutboundHandler 接口，继承 ChannelHandler 接口。

流程如下图：



ChannelInBoundHandler 负责数据进入并在 ChannelPipeline 中按照从上至下的顺序查找调用相应的 InBoundHandler。

ChannelOutBoundHandler 负责数据出去并在 ChannelPipeline 中按照从下至上的顺序查找调用相应的 OutBoundHandler。

2.在 5.x 版本中的改动

在 5.x 版本中。作者再次对 ChannelHandler 进行了改动。

在更新说明里可以看到：

作者简化了 Handler 的类型层次结构。

ChannelInboundHandler 和 ChannelOutboundHandler 接口合并到 ChannelHandler 里面。

ChannelInboundHandlerAdapter，ChannelOutboundHandlerAdapter 以及 ChannelDuplexHandlerAdapter 被取消，其功能被 ChannelHandlerAdapter 代替。

由于上述的改动，开发者将无法区分 InBoundHandler 和 OutBoundHandler 所以 CombinedChannelDuplexHandler 的功能也被 ChannelHandlerAdapter 代替。

有兴趣了解的可以看一下注释。假如不看也影响不大。因为 5.x 看上去改动很大，实际上框架的设计思路并没有改变。

(注释：

5.x 版本中虽然删除了 InBoundHandler 和 OutBoundHandler，但是在设计思想上 InBound 和 OutBound 的概念还是存在的。只不过是作者使用了另外一种方式去实现罢了。

查看过 4.x 版本代码的朋友可能已经了解知道了。消息在管道中都是以 ChannelHandlerContext 的形势传递的。而 InBound 和 OutBound 主要作用是当被当做 ChannelPipeline 管道中标识。用于 Handler 中相对应的调用处理，通过两个布尔值变量 inBound 和 outBound 来区分是进入还是出去。并以此来区分 Handler 并调用相应的方法，其实没有什么实际用途。于是作者在 5.x 版本中对此做出了优化。优化方案笔者感觉 very nice。

由于删除了 InBoundHandler 和 OutBoundHandler 的接口。作者在 DefaultChannelHandlerContext 中重写了 findContextInBound()和 findContextOutBound()方法。并且在方法里引入了参数 mask。

在类开始处定义静态终态的变量来标记 4.x 版本中定义的 InBound 和 OutBound 中的方法名(可以变相的认为是枚举)。在源代码中的实现是利用 mask 来获取对应的 flag，最终实现使用 mask 来区分 InBoundHandler 亦或是 OutBoundHandler。

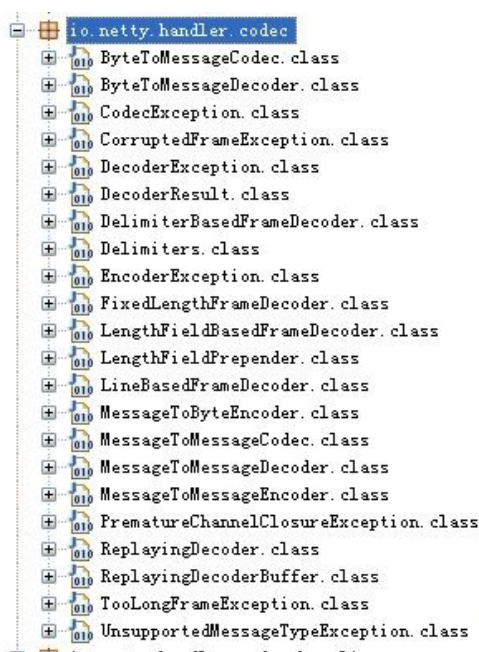
这样的改动，优点显而易见。简化了层次结构，降低了框架的复杂度。同时功能上却没有有什么变化。易于使用了解。

)

目前在不涉及框架底层的情况下。笔者将继续使用 Netty 4.0.14 final 版本。正如第一章介绍所说。5.x 版本作者并没有进行大规模的设计变更。仅仅只是局部的小部分修改。所以在版本没有稳定之前。教程都将采用 4.0.14final 为框架包。

3.认识 Handler 中的编解码器

编解码器在 Netty 框架中占了相当大的一部分代码量。由此可见其重要性。本章内容旨在阐述编解码器的基础。下一章会详细分类的按照框架的结构详解其余编解码器。



在 Netty 的 codec 包内部我们可以看到很多的编解码器和一些异常捕获。

Netty4.x 中文教程系列(五)编解码器 Codec

这篇文章主要在于讲述 Handler 里面的 Codec，也就是相关的编解码器。原本想把编解码器写在上一篇文章里面的。后来想想 Netty 里面的编解码器太多了。想要一次写完比较困难。于是重新开了一篇文章来专门写这个。

1. Hello World！实例中的使用

在这里先讲一下我们第一篇文章里面的实例使用到编解码器。

1.1 DelimiterBasedFrameDecoder 解码器

DelimiterBasedFrameDecoder 顾名思义我们可以理解为基于分隔符的帧解码器。参数有两个，一个是最大帧长度，另外一个定义分隔符。

在 Delimiters 中提供给我们两种分隔符。一种是“0x00-NUL”分隔符。另外一种就是实例中使用的“\r\n”或“\n”分隔符。

```
public DelimiterBasedFrameDecoder(
    int maxFrameLength, boolean stripDelimiter, boolean failFast, ByteBuf... delimiters) {
    validateMaxFrameLength(maxFrameLength);
    if (delimiters == null) {
        throw new NullPointerException("delimiters");
    }
    if (delimiters.length == 0) {
        throw new IllegalArgumentException("empty delimiters");
    }

    if (isLineBased(delimiters) && !isSubclass()) {
        lineBasedDecoder = new LineBasedFrameDecoder(maxFrameLength, stripDelimiter, failFast);
        this.delimiters = null;
    } else {
        this.delimiters = new ByteBuf[delimiters.length];
        for (int i = 0; i < delimiters.length; i++) {
            ByteBuf d = delimiters[i];
            validateDelimiter(d);
            this.delimiters[i] = d.slice(d.readerIndex(), d.readableBytes());
        }
        lineBasedDecoder = null;
    }
    this.maxFrameLength = maxFrameLength;
    this.stripDelimiter = stripDelimiter;
    this.failFast = failFast;
}
```

在构造函数中我们可以看出，当分隔符是“\n”的时候，框架默认解码器为基于行的帧解码器（LineBasedFrameDecoder）。否则按照可读取比特长度进行帧解码。

1.2 StringDecoder 字符串解码器 和 编码器

解码器:将比特流转换为默认编码的字符串。默认编码为 UTF-8。当然开发者可以通过设置字符编码参数来设置字符编码。编码器:将字符串转换为 Byte[]

2. Netty 中 Handler 详述

在 Netty 的类库的 handler 目录可以看出它的基本结构(下图)：

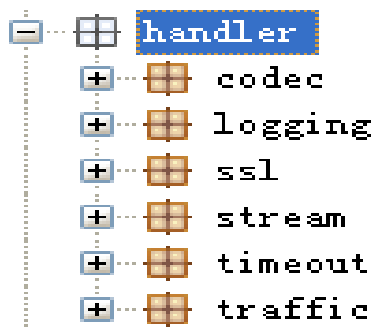


图 2.1 handler 包结构

整个包由 6 个主要部分组成，笔者将由简入繁，慢慢想读者解释每个包的含义和用法。(若有不正确之处，希望大家能给予指点)

2.1 Logging 日志



用于 Netty 中的日志输出。

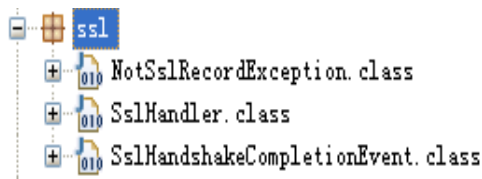
2.1.1 loggingHandler

LoggingHandler 继承于 ChannelDuplexHandler。它的注释我们可以看出：“是一个使用日志框架记录全部事件的 ChannelHandler，缺省值是记录全部 DEBUG 级别以上的事件”。它的功能是记录全部事件，包含 Inbound 和 Outbound 的，之所以选择了继承 ChannelDuplexHandler，是由于 ChannelDuplexHandler 继承 ChannelInboundHandlerAdapter 实现 ChannelOutboundHandler。所以相当于 Netty 框架内的全部通信相关的事件都会得到处理。

2.1.2 LogLevel

在这里作者定义了 5 个级别的 log。TRACE，DEBUG，INFO，WARN，ERROR。

2.2 Ssl



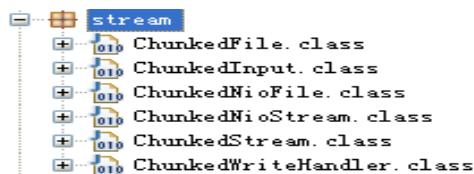
用于 SSL 协议解析和编码。

2.2.1 SslHandler

熟悉了解过 Http 的朋友应该是知道 ssl 的。SSL 的英文全称是 “Secure Sockets Layer”，中文名为 “安全套接层协议”，它是网景（Netscape）公司提出的基于 WEB 应用的安全协议。SSL 协议指定了一种在应用程序协议（如 HTTP、Telenet、NMTP 和 FTP 等）和 TCP/IP 协议之间提供数据安全性分层的机制，它为 TCP/IP 连接提供数据加密、服务器认证、消息完整性以及可选的客户机认证。现在的相当一部分网站都有 SSL 加密。而 SslHandler 则是 Netty 提供的 Ssl 解码编码处理。

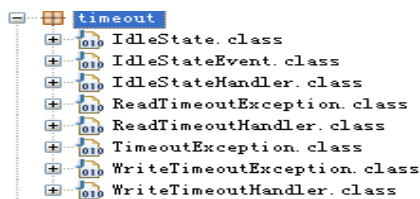
2.2.2 NotSslRecordException 和 SslHandshakeCompletionEvent 抛出异常和处理完成触发事件。

2.3 Stream 流



用于文件的的传输。将 Java 里面的 File 转换为 Stream 流，然后进行传输。

2.4 Timeout 空闲检测



用于 Netty 框架中空闲超时相关。

IdleState 空闲状态。Netty 中的空闲时间包括，读空闲，写空闲和读写空闲 3 种时间。

顾名思义，读空闲即一段时间内没有接受到消息，写空闲即一段时间内没有发送消息。读写空闲即一段时间内读写都空闲。主要是用于检测空闲状态。并且特定条件下服务端关闭和释放一些资源。

包 compression : 用于 ByteBuf 数据压缩和解压缩的。继承 MessageToMessageEncoder<ByteBuf>

包 http : 用于 HTTP 请求相关的。这个包里面就比较复杂了。下面详细讲一下。

1.包空间 : HTTP 内容, 请求, 响应等。

2.包 cors: 包名称是 (跨域资源分享) Cross Origin Resource Sharing 的简写。用于客户端跨域请求。 可以参考 <http://www.w3.org/TR/cors/>

3.包 multipart : POST 消息和文件上传相关的一些。只是粗略看了一下。

4.包 websocketx : 针对近年来 Html5 发展起来的 websocket 技术的。不过貌似由于 Html5 标准还未正式的确定。所以这个包里面的内容比较多。编解码器版本也很多。相信以后统一标准之后会简单一些。暂时不推荐吧

包 marshalling :

包 protoBuf : 用于 [Google Protocol Buffers](#) 编解码

包 rtsp : 实时流传输协议 (Real Time Streaming Protocol , RTSP)

包 sctp : 流控制传输协议 (Stream Control Transmission Protocol , SCTP)

包 serialization : 用于序列化的 Java 对象的和 ByteBuf 之间的转换。

包 socks : 用于 Java Socket 通信相关的。支持 Socket4a 和 Socket 5 两个版本

包 spdy : SPDY 协议是近年来发展的一种协议。主要目的是为了减少网页加载的时间。它是 HTTP 协议的增强版本。它从某种程度上讲提高了 HTTP 协议在数据传输时的速度和性能

包 string : 用于 java 里面字符串的编解码

下面开始正文 :

纵览 Netty 框架的包结构, 不难看出。其实 Netty 是有五大模块组成。



- 1.Bootstrap 负责启动引导
- 2.Buffer 是 Netty 自己封装的缓存器
- 3.Channel 负责管理和建立连接
- 4.Handler 是责任链路模式中的处理者
- 5.Util 是 Netty 提供和使用到的一些工具

如何启动 Netty 服务器

Netty 的启动服务器相关的类全部都在 bootstrap 包里面。所以本章我们从头开始, 从 bootstrap 包里面的内容开始。从创建一个 Netty 服务器开始为大家逐步讲解 Netty 的应用。

相比于第五章的 ChannelHandler 里面的编解码器 bootstrap 里面可以说是内容少的可怜。来看一下他的包内容：



简简单单的三个类，一个接口。

Bootstrap 是客户端的启动程序类。

ServerBootstrap 是服务端的启动程序类

Bootstrap 和 ServerBootstrap 继承 AbstractBootstrap。

ChannelFactory 则是 AbstractBootstrap 中用于创建 Channel 的接口

以下代码以服务端的启动程序启动为例：

步骤一：实例化 ServerBootstrap

首先我们需要实例化一个 ServerBootstrap 服务端启动引导程序。如下图：

```
ServerBootstrap bootstrap = new ServerBootstrap();
```

步骤二：设置它的线程组

创建两个 NioEventLoopGroup，一个是父线程（Boss 线程），一个是子线程(work 线程)。

```
NioEventLoopGroup parentBosser = new NioEventLoopGroup();  
NioEventLoopGroup childWorker = new NioEventLoopGroup();
```

设置 bootstrap 的线程组

```
bootstrap.group(parentBosser, childWorker);
```

设置线程组主要的目的是为了处理 Channel 中的事件和 IO 操作。

下图为 ServerBootstrap 的 group 方法的源码：

```
public ServerBootstrap group(EventLoopGroup parentGroup, EventLoopGroup childGroup) {  
    super.group(parentGroup);  
    if (childGroup == null) {  
        throw new NullPointerException("childGroup");  
    }  
    if (this.childGroup != null) {  
        throw new IllegalStateException("childGroup set already");  
    }  
    this.childGroup = childGroup;  
    return this;  
}
```

父线程组被传递到父类中。详细的解释在最后面。涉及的东西太多。在后面在进行解释。

步骤三：设置 Channel 类型

设置 Channel 类型：

```
bootstrap.channel(NioServerSocketChannel.class);
```

下图 ServerBootstrap 中 channel()方法的源码：

```
public B channel(Class<? extends C> channelClass) {  
    if (channelClass == null) {  
        throw new NullPointerException("channelClass");  
    }  
    return channelFactory(new BootstrapChannelFactory<C>(channelClass));  
}
```

我们可以看到创建并设置了一个 Channel 工厂。

下图是 BootstrapChannelFactory 的源码。它是一个终态的静态的类。实现 ChannelFactory。作用是根
据初始设置的 Channel 类型，创建并返回一个新的 Channel。


```

private static final class BootstrapChannelFactory<T extends Channel> implements ChannelFactory<T> {
    private final Class<? extends T> clazz;

    BootstrapChannelFactory(Class<? extends T> clazz) {
        this.clazz = clazz;
    }

    @Override
    public T newChannel() {
        try {
            return clazz.newInstance();
        } catch (Throwable t) {
            throw new ChannelException("Unable to create Channel from class " + clazz, t);
        }
    }

    @Override
    public String toString() { return StringUtil.simpleClassName(clazz) + ".class"; }
}

```

步骤四：设置责任链路

责任链模式是 Netty 的核心部分。每个处理器只负责自己有关的东西。然后将处理结果根据责任链传递下去。

```

bootstrap.childHandler(new ChannelInitializer<NioSocketChannel>() {

```

我们要在初始的设置一个责任链路。当一个 Channel 被创建之后初始化的时候将被设置。下图是 ServerBootstrap 在 init() 方法的源码：

```

final EventLoopGroup currentChildGroup = childGroup;
final ChannelHandler currentChildHandler = childHandler;
final Entry<ChannelOption<?>, Object>[] currentChildOptions;
final Entry<AttributeKey<?>, Object>[] currentChildAttrs;
synchronized (childOptions) {
    currentChildOptions = childOptions.entrySet().toArray(newOptionArray(childOptions.size()));
}
synchronized (childAttrs) {
    currentChildAttrs = childAttrs.entrySet().toArray(newAttrArray(childAttrs.size()));
}

p.addLast(new ChannelInitializer<Channel>() {
    @Override
    public void initChannel(Channel ch) throws Exception {
        ch.pipeline().addLast(new ServerBootstrapAcceptor(
            currentChildGroup, currentChildHandler, currentChildOptions, currentChildAttrs));
    }
});

```

创建一个 Channel，在初始化的设置管道里面的处理器。

步骤五：绑定并监听端口

绑定并设置监听端口。

```
try {
    // 绑定并监听端口
    ChannelFuture future = bootstrap.bind(9002).sync();
    // 等待关闭事件
    future.channel().closeFuture().sync();
} catch (InterruptedException e) {
    throw new Exception("");
} finally {
    // 释放资源
    parentBossor.shutdownGracefully();
    childWorker.shutdownGracefully();
}
```

经过以上的 5 个步骤，我们的服务器就足以启动了。很多的设置都是 Netty 默认的。我们想设置自己的参数怎么办呢？Netty 提供了这个方法。

步骤六：其他设置

1. 设置 Channel 选项配置：

在 Netty 以前的版本中都是以字符串来配置的。4.x 版本发布之后统一修改为使用 ChannelOption 类来实现配置。

例如：

```
bootstrap.option(ChannelOption.SO_KEEPALIVE, true);
```

Socket 连接是否保存连接：

```
public static final ChannelOption<Boolean> SO_KEEPALIVE = valueOf("SO_KEEPALIVE");
```

还有很多其他的参数。如下图所示：

```
ALLOCATOR: ChannelOption<ByteBufAllocator> = valueOf(...)
ALLOW_HALF_CLOSURE: ChannelOption<Boolean> = valueOf(...)
AUTO_CLOSE: ChannelOption<Boolean> = valueOf(...)
AUTO_READ: ChannelOption<Boolean> = valueOf(...)
CONNECT_TIMEOUT_MILLIS: ChannelOption<Integer> = valueOf(...)
DATAGRAM_CHANNEL_ACTIVE_ON_REGISTRATION: ChannelOption<Boolean> = valueOf(...)
IP_MULTICAST_ADDR: ChannelOption<InetAddress> = valueOf(...)
IP_MULTICAST_IF: ChannelOption<NetworkInterface> = valueOf(...)
IP_MULTICAST_LOOP_DISABLED: ChannelOption<Boolean> = valueOf(...)
IP_MULTICAST_TTL: ChannelOption<Integer> = valueOf(...)
IP_TOS: ChannelOption<Integer> = valueOf(...)
MAX_MESSAGES_PER_READ: ChannelOption<Integer> = valueOf(...)
MESSAGE_SIZE_ESTIMATOR: ChannelOption<MessageSizeEstimator> = valueOf(...)
pool: ConstantPool<ChannelOption<Object>> = new ConstantPool<ChannelOption<Ob
RCVBUF_ALLOCATOR: ChannelOption<RecvByteBufAllocator> = valueOf(...)
SO_BACKLOG: ChannelOption<Integer> = valueOf(...)
SO_BROADCAST: ChannelOption<Boolean> = valueOf(...)
SO_KEEPALIVE: ChannelOption<Boolean> = valueOf(...)
SO_LINGER: ChannelOption<Integer> = valueOf(...)
SO_RCVBUF: ChannelOption<Integer> = valueOf(...)
SO_REUSEADDR: ChannelOption<Boolean> = valueOf(...)
SO_SNDBUF: ChannelOption<Integer> = valueOf(...)
SO_TIMEOUT: ChannelOption<Integer> = valueOf(...)
TCP_NODELAY: ChannelOption<Boolean> = valueOf(...)
WRITE_BUFFER_HIGH_WATER_MARK: ChannelOption<Integer> = valueOf(...)
WRITE_BUFFER_LOW_WATER_MARK: ChannelOption<Integer> = valueOf(...)
WRITE_SPIN_COUNT: ChannelOption<Integer> = valueOf(...)
```

这里不详细讲了。参考：[io.netty.channel.ChannelOption](#)

2. 设置子 Channel 的属性：



设置子 Channel 的属性。当值为 null 是，属性将被删除。

解释 EventLoopGroup

这里解释一下我们上面创建的两个完全一样的线程组的作用。

Netty 的架构使用了非常复杂的主从式 Reactor 线程模型。简单的说就是。父线程组（代码中的 parentBoss）担任（acceptor）的角色。负责接收客户端的连接请求，处理完成请求，创建一个 Channel 并注册到子线程组（代码中的 childWorker）中的某个线程上面，然后这个线程将负责 Channel 的读写，编解码等操作。

源代码查看：

在步骤四中我们设置了责任链路。这里是 Channel 初始化和注册。在这里的 init 就是 Channel 的初始化。初始化完成之后。Group()则是获取在步骤一种的设置父线程组，并将这个新的 Channel 注册进来。

下图是 AbstractBootstrap 的 initAndRegister 方法

```

final ChannelFuture initAndRegister() {
    final Channel channel = channelFactory().newChannel();
    try {
        init(channel);
    } catch (Throwable t) {
        channel.unsafe().closeForcibly();
        // as the Channel is not registered yet we need to force the usage of the GlobalEventExecutor
        return new DefaultChannelPromise(channel, GlobalEventExecutor.INSTANCE).setFailure(t);
    }

    ChannelFuture regFuture = group().register(channel);
    if (regFuture.cause() != null) {
        if (channel.isRegistered()) {
            channel.close();
        } else {
            channel.unsafe().closeForcibly();
        }
    }

    // If we are here and the promise is not failed, it's one of the following cases:
    // 1) If we attempted registration from the event loop, the registration has been completed at this point.
    //    i.e. It's safe to attempt bind() or connect() now because the channel has been registered.
    // 2) If we attempted registration from the other thread, the registration request has been successfully
    //    added to the event loop's task queue for later execution.
    //    i.e. It's safe to attempt bind() or connect() now:
    //        because bind() or connect() will be executed *after* the scheduled registration task is executed
    //        because register(), bind(), and connect() are all bound to the same thread.

    return regFuture;
}

```

方法 Init()实现在 ServerBootstrap 中。代码如下：

```

void init(Channel channel) throws Exception {
    final Map<ChannelOption<?>, Object> options = options();
    synchronized (options) {
        channel.config().setOptions(options);
    }

    final Map<AttributeKey<?>, Object> attrs = attrs();
    synchronized (attrs) {
        for (Entry<AttributeKey<?>, Object> e: attrs.entrySet()) {
            //unchecked/
            AttributeKey<Object> key = (AttributeKey<Object>) e.getKey();
            channel.attr(key).set(e.getValue());
        }
    }

    ChannelPipeline p = channel.pipeline();
    if (handler() != null) {
        p.addLast(handler());
    }

    final EventLoopGroup currentChildGroup = childGroup;
    final ChannelHandler currentChildHandler = childHandler;
    final Entry<ChannelOption<?>, Object>[] currentChildOptions;
    final Entry<AttributeKey<?>, Object>[] currentChildAttrs;
    synchronized (childOptions) {
        currentChildOptions = childOptions.entrySet().toArray(newOptionArray(childOptions.size()));
    }
    synchronized (childAttrs) {
        currentChildAttrs = childAttrs.entrySet().toArray(newAttrArray(childAttrs.size()));
    }

    p.addLast(new ChannelInitializer<Channel>() {
        @Override
        public void initChannel(Channel ch) throws Exception {
            ch.pipeline().addLast(new ServerBootstrapAcceptor(
                currentChildGroup, currentChildHandler, currentChildOptions, currentChildAttrs));
        }
    });
}

```

看到下面的代码是不是有种和熟悉的感觉？没错。就是在步骤四中设置责任链路的那段代码。这里将注册新创建的 Channel 到子线程组

```

import io.netty.bootstrap.ServerBootstrap;
import io.netty.channel.*;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.nio.NioServerSocketChannel;
import io.netty.channel.socket.nio.NioSocketChannel;
import io.netty.handler.codec.LengthFieldBasedFrameDecoder;

```

```

import io.netty.handler.codec.LengthFieldPrepender;

/**
 * 测试。。O(∩_∩)O 哈哈~
 * Created by TinyZ on 2014/8/12.
 */
public class MainTest {

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

        NioEventLoopGroup parentBosser = new NioEventLoopGroup();
        NioEventLoopGroup childWorker = new NioEventLoopGroup();

        ServerBootstrap bootstrap = new ServerBootstrap();
        bootstrap.group(parentBosser, childWorker);
        bootstrap.channel(NioServerSocketChannel.class);
        bootstrap.childHandler(new ChannelInitializer<NioSocketChannel>() {
            @Override
            protected void initChannel(NioSocketChannel ch) throws Exception {
                ChannelPipeline cp = ch.pipeline();
                // 基于长度的解码器
                cp.addLast("framer", new LengthFieldBasedFrameDecoder(Integer.MAX_VALUE, 0, 2,
0, 2));
                cp.addLast("prepender", new LengthFieldPrepender(4));
                //
                cp.addLast("handler", new SimpleChannelInboundHandler<Object>() {

                    @Override
                    protected void channelRead0(ChannelHandlerContext ctx, Object msg) throws
Exception {

                        System.out.println();
                        ctx.channel().writeAndFlush(msg);

                    }
                });
            }
        });
        bootstrap.option(ChannelOption.SO_KEEPALIVE, true);
        //bootstrap.childAttr()
        try {
            // 绑定并监听端口
            ChannelFuture future = bootstrap.bind(9002).sync();
            // 等待关闭事件
            future.channel().closeFuture().sync();
        } finally {
            // 释放资源
            parentBosser.shutdownGracefully();
            childWorker.shutdownGracefully();
        }
    }
}

```

1. 构建 UDP 服务端

首先我们应该清楚 UDP 协议是一种无连接状态的协议。所以 Netty 框架区别于一般的有链接协议服务端启动程序 (ServerBootstrap)。

Netty 开发基于 UDP 协议的服务端需要使用 Bootstrap

```
package dev.tinyz.game;

import io.netty.bootstrap.Bootstrap;
import io.netty.buffer.Unpooled;
import io.netty.channel.*;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.DatagramPacket;
import io.netty.channel.socket.nio.NioDatagramChannel;
import io.netty.handler.codec.MessageToMessageDecoder;

import java.net.InetSocketAddress;
import java.nio.charset.Charset;
import java.util.List;

/**
 * @author TinyZ on 2015/6/8.
 */
public class GameMain {

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

        final NioEventLoopGroup nioEventLoopGroup = new NioEventLoopGroup();

        Bootstrap bootstrap = new Bootstrap();
        bootstrap.channel(NioDatagramChannel.class);
        bootstrap.group(nioEventLoopGroup);
        bootstrap.handler(new ChannelInitializer<NioDatagramChannel>() {

            @Override
            public void channelActive(ChannelHandlerContext ctx) throws Exception {
                super.channelActive(ctx);
            }

            @Override
            protected void initChannel(NioDatagramChannel ch) throws Exception {
                ChannelPipeline cp = ch.pipeline();
                cp.addLast("framer", new MessageToMessageDecoder<DatagramPacket>() {
                    @Override
                    protected void decode(ChannelHandlerContext ctx, DatagramPacket msg,
List<Object> out) throws Exception {
                        out.add(msg.content().toString(Charset.forName("UTF-8")));
                    }
                });
            }
        });
    }
}
```

```

        }
    }).addLast("handler", new UdpHandler());
}
});
// 监听端口
ChannelFuture sync = bootstrap.bind(9009).sync();
Channel udpChannel = sync.channel();

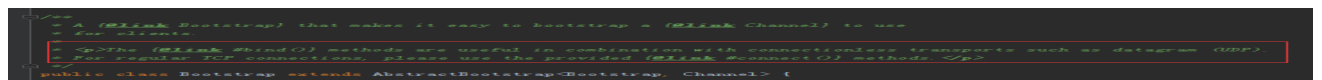
// String data = "我是大好人啊";
// udpChannel.writeAndFlush(new
DatagramPacket(Unpooled.copiedBuffer(data.getBytes(Charset.forName("UTF-8"))), new
InetSocketAddress("192.168.2.29", 9008)));

Runtime.getRuntime().addShutdownHook(new Thread(new Runnable() {
    @Override
    public void run() {
        nioEventLoopGroup.shutdownGracefully();
    }
}));
}
}
}

```

于 Tcp 协议的客户端启动程序基本一样。唯一区别就在于，UDP 服务器使用的是 bind 方法，来监听端口

在 Netty 的 Bootstrap 类中的注释，发现有如下注释内容：



```

// A Bootstrap that makes it easy to bootstrap a Channel to use
// for a Channel.
// The Bootstrap methods are useful in conjunction with ChannelFactory objects such as DatagramChannelFactory.
// See ChannelFactory for more details.
public class Bootstrap extends AbstractBootstrap<Bootstrap, Channel> {

```

大意就是：bind()用于 UDP，TCP 连接使用 connect()。

上面的源码监听的是端口 9009，那么所有使用 UDP 协议的数据，发送到端口 9009，就会被我们的 Netty 接收到了。

为了输出方便，博主在上面的代码中增加一个 MessageToMessageDecoder 将接收到的 Datagram，排除其他信息，仅将字符串传递下去。并在 UdpHandler 中打印出来。

2. 构建 UDP 客户端

UDP 协议来说，其实没有客户端和服务端的区别啦。只是为了贴近 TCP 协议做的一点文字描述上面的区分。

简单来讲，上面的那段逻辑其实就可以作为 UDP 客户端来使用。注释掉的那行逻辑其实就是发送“我是大好人啊”这个字符串到 ip 地址为 192.168.2.29 的服务端的 9008 端口。代码如下：

```

package dev.tinyz.game;

import io.netty.bootstrap.Bootstrap;
import io.netty.buffer.Unpooled;

```

```

import io.netty.channel.*;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.DatagramPacket;
import io.netty.channel.socket.nio.NioDatagramChannel;
import io.netty.handler.codec.MessageToMessageDecoder;

import java.net.InetSocketAddress;
import java.nio.charset.Charset;
import java.util.List;

/**
 * @author TinyZ on 2015/6/8.
 */
public class GameMain {

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

        final NioEventLoopGroup nioEventLoopGroup = new NioEventLoopGroup();

        Bootstrap bootstrap = new Bootstrap();
        bootstrap.channel(NioDatagramChannel.class);
        bootstrap.group(nioEventLoopGroup);
        bootstrap.handler(new ChannelInitializer<NioDatagramChannel>() {

            @Override
            public void channelActive(ChannelHandlerContext ctx) throws Exception {
                super.channelActive(ctx);
            }

            @Override
            protected void initChannel(NioDatagramChannel ch) throws Exception {
                ChannelPipeline cp = ch.pipeline();
                cp.addLast("framer", new MessageToMessageDecoder<DatagramPacket>() {
                    @Override
                    protected void decode(ChannelHandlerContext ctx, DatagramPacket msg,
List<Object> out) throws Exception {
                        out.add(msg.content().toString(Charset.forName("UTF-8")));
                    }
                }).addLast("handler", new UdpHandler());
            }
        });
        // 监听端口
        ChannelFuture sync = bootstrap.bind(0).sync();
        Channel udpChannel = sync.channel();

        String data = "我是大好人啊";
        udpChannel.writeAndFlush(new
DatagramPacket(Unpooled.copiedBuffer(data.getBytes(Charset.forName("UTF-8"))), new
InetSocketAddress("192.168.2.29", 9008)));

        Runtime.getRuntime().addShutdownHook(new Thread(new Runnable() {
            @Override

```



```

        public void run() {
            nioEventLoopGroup.shutdownGracefully();
        }
    }));
}
}
}

```

和上面的“服务端”代码最大的差别就是，监听的端口号修改成 0.为

使用 Netty 的 Channel 发送 DatagramPacket。写好目标地址，然后运行起来就可以自己测试一下了。

3. JAVA 原生 UDP

有朋友这个时候就会问：为什么不是有 JAVA 原生的 UDP 呢？

其实很简单。说白了 Netty 使用的也是 Java 底层的代码。只是做了一层封装，以便于使用。服务端使用 Netty 框架构建高性能，高扩展的 UDP 服务器。

客户端则使用 JAVA 或者任意其他的语言的 API（遵循 UDP 协议即可）。

下面上一段博主使用的的 JAVA

```

package dev.tinyz.game;

import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetSocketAddress;
import java.nio.charset.Charset;

/**
 * @author TinyZ on 2015/6/10.
 */
public class UdpTest {

    public static void main(String[] args) throws IOException {
        final String data = "博主邮箱:zou90512@126.com";
        byte[] bytes = data.getBytes(Charset.forName("UTF-8"));
        InetSocketAddress targetHost = new InetSocketAddress("192.168.2.29", 9009);

        // 发送 udp 内容
        DatagramSocket socket = new DatagramSocket();
        socket.send(new DatagramPacket(bytes, 0, bytes.length, targetHost));
    }
}

```

ps.UDP 协议最大特点就是效率高，速度快。用于某些场合可以极大改善系统的性能。

博主在这里引入这个 Netty 实现 UDP 的服务端，主要目的。嘻嘻。就是想开源拙作：eyeOfSauron 日志系统。