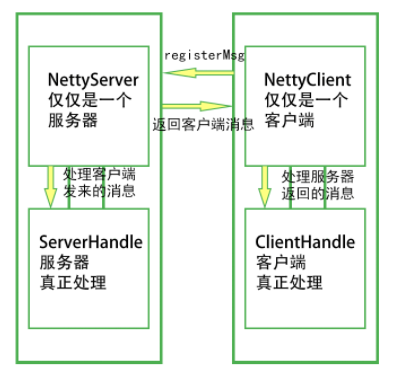
Netty 用户指南

Netty是一个提供异步事件驱动的网络应用框架，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，其是一个NIO（Non-block IO）框架，使用它可以简单快速开发网络应用程序。Netty通信的整体图如下所示：



# 1.Discard服务

下面介绍一个简单的例子：Discard服务用于丢弃所有接受到的数据，不做有任何有响应的协议，首先从Handler的实现开始，Handler用于Netty产生的I/O事件，代码如下：

*package io.netty.example.discard;*

*import io.netty.buffer.ByteBuf;*

*import io.netty.channel.ChannelHandlerContext;*

*import io.netty.channel.ChannelInboundHandlerAdapter;*

*public class DiscardServerHandler extends ChannelInboundHandlerAdapter { // (1)*

*@Override*

*public void channelRead(ChannelHandlerContext ctx, Object msg) { // (2)*

*// Discard the received data silently.*

*((ByteBuf) msg).release(); // (3)*

*}*

*@Override*

*public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) { // (4)*

*// Close the connection when an exception is raised.*

*cause.printStackTrace();*

*ctx.close();*

*}*

*}*

1. DiscardServerHandler继承自ChannelInboundHandlerAdapter，实现了接口ChannelIn

boundHandler。ChannelInboundHandler提供多种可重写的事件处理器。

1. 示例上重写了channelRead事件处理器方法，当服务端接收到客户端发送过来的数据时，调用该方法处理接收到消息。接收的数据类型是ByteBuf。
2. Discard协议的实现，处理器忽略接收到数据，ByteBuf是一个引用计数对象，通过release方法显示的释放掉，通常channelRead处理器方法的实现如下：

*public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {*

*ctx.fireChannelRead(msg);*

*}*

4）exceptionCaught事件处理是当出现Throwable对象才会被调用，当Netty I/O及handler事件处理出现异常时，记录捕获的异常并关闭channel，该方法用于在关闭之前进行处理，发送错误响应信息。

以上已经实现了Discard服务的前部分工作，下面是通过main方法来启动DiscardServerHandler，源码如下：

*package io.netty.example.discard;*

*import io.netty.bootstrap.ServerBootstrap;*

*import io.netty.channel.ChannelFuture;*

*import io.netty.channel.ChannelInitializer;*

*import io.netty.channel.ChannelOption;*

*import io.netty.channel.EventLoopGroup;*

*import io.netty.channel.nio.NioEventLoopGroup;*

*import io.netty.channel.socket.SocketChannel;*

*import io.netty.channel.socket.nio.NioServerSocketChannel;*

*public class DiscardServer {*

*private int port;*

*public DiscardServer(int port) {*

*this.port = port;*

*}*

*public void run() throws Exception {*

*EventLoopGroup bossGroup = new NioEventLoopGroup(); // (1)*

*EventLoopGroup workerGroup = new NioEventLoopGroup();*

*try {*

*ServerBootstrap b = new ServerBootstrap(); // (2)*

*b.group(bossGroup, workerGroup)*

*.channel(NioServerSocketChannel.class) // (3)*

*.childHandler(new ChannelInitializer<SocketChannel>() { // (4)*

*@Override*

*public void initChannel(SocketChannel ch) throws Exception {*

*ch.pipeline().addLast(new DiscardServerHandler());*

*}*

*})*

*.option(ChannelOption.SO\_BACKLOG, 128) // (5)*

*.childOption(ChannelOption.SO\_KEEPALIVE, true); // (6)*

*// Bind and start to accept incoming connections.*

*ChannelFuture f = b.bind(port).sync(); // (7)*

*// Wait until the server socket is closed.*

*// In this example, this does not happen, but you can do that to gracefully*

*// shut down your server.*

*f.channel().closeFuture().sync();*

*} finally {*

*workerGroup.shutdownGracefully();*

*bossGroup.shutdownGracefully();*

*}*

*}*

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

*int port;*

*if (args.length > 0) {*

*port = Integer.parseInt(args[0]);*

*} else {*

*port = 8080;*

*}*

*new DiscardServer(port).run();*

*}*

*}*

1. NioEventLoopGroup是处理I/O操作的多线程事件循环器，Netty为不同的传输协议提供不同的EventLoopGroup。实现Server端的程序，需要使用两个NioEventLoopGroup，第一个常称为boss，用来接收进来的连接，第二个常被叫做worker，用来处理已经被接收到的连接。如何知道多少个线程已经被使用，如何映射到已经创建的Channels上都需要依赖EventLoopGroup的实现，并且可以通过构造函数来配置他们的关系。
2. ServerBootstrap，是启动NIO 服务的辅助启动类，也可以直接使用channel，但是过程比较负责
3. 使用NioServerSocketChannel类来为新连接建立一个Channel
4. 这里定义的handler用来处理最近接收的Channel，ChannelInitializer是一个特殊处理类，帮助使用者配置一个新的Channel。最常用的是配置ChannlePipeline增加DiscardServerHandler的Channel来处理业务，如果应用比较复杂可以在pipeline中添加多handlers。
5. 在Channel实现的时候可以配置参数，例如TCP/IP server，可以设置socket option，例如tcpNoDealy及keepAlive等。示例中，option()是提供给NioServerSocketChannel用来接收进来的链接，childOption是提供给父管道ServerChannel接收到的连接，也就是示例中定义的NioServerSocketChannel
6. 绑定端口并启动，也可以多次调用bind方法，绑定不同的地址

运行程序后，可以通过telnet localhost 8080，连接到Netty Server，为了打印输出，将channelRead方法修改如下：

*@Override*

*public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {*

*ByteBuf in = (ByteBuf)msg;*

*byte[] req = new byte[in.readableBytes()];*

*in.readBytes(req);*

*String body = new String(req, "UTF-8");*

*System.out.println("Received Message:" + body);*

*}*

Server端，输出如下：

*Received Message:first mesg*

Discard服务中，虽然接收到数据，但是没有做任何的响应，ECHO例子中针对接收到的数据返回一个响应，与Discard对比，不同的地方是channelRead，如下所示：

*@Override*

*public void channelRead(ChannelHandlerContext ctx, Object msg) {*

*ctx.write(msg); // (1)*

*ctx.flush(); // (2)*

*}*

1. ChannelHandlerContext对象提供许多操作，能够触发各种I/O事件和操作，上例中调用write(Object)方法逐字的把接收到的消息写入
2. ctx.write(Object)方法不会使消息写入到通道上，而是缓冲在内部，需要调用ctx.flush方法把缓冲区的数据强行输出，可以调用ctx.writeAndFlush(msg)达到同样的目的。

# 2.Spark Netty Demo

Spark在2.2以后，就使用Netty作为Rpc的底层框架，下面是用Scala实现的Netty demo

## 2.1 Server Handler

*class ServerHandler extends ChannelInboundHandlerAdapter {*

*//客户端建立连接后调用*

*override def channelActive(ctx: ChannelHandlerContext): Unit ={*

*println("Channel Active invoked")*

*}*

*//接收客户端发送过来的消息*

*override def channelRead(ctx: ChannelHandlerContext, msg: Any): Unit = {*

*println("channelRead invoked")*

*val byteBuf = msg.asInstanceOf[ByteBuf]*

*val bytes = new Array[Byte](byteBuf.readableBytes())*

*byteBuf.readBytes(bytes)*

*val message = new String(bytes, "UTF-8")*

*println(message)*

*val back = "Good boy"*

*val resp = Unpooled.copiedBuffer(back.getBytes("UTF-8"))*

*println(msg)*

*ctx.write(resp)*

*}*

*//将消息队列中的数据写入SocketChannel并发送给对方*

*override def channelReadComplete(ctx: ChannelHandlerContext): Unit = {*

*println("Channel ReadComplete invoked")*

*ctx.flush()*

*}*

*}*

## 2.2 Server

*class NettyServer {*

*def bind(host: String, port: Int): Unit = {*

*//配置服务端线程池组，用户服务端接收客户端连接*

*val bossGroup = new NioEventLoopGroup()*

*//用户进行SocketChannel的网络读写*

*val workerGroup = new NioEventLoopGroup()*

*try { //Netty用户启动NIO服务端的辅助启动类，降低服务端开发复杂度*

*val bootstrap = new ServerBootstrap()*

*//将两个NIO线程组作为参数传入到ServerBootStrap*

*bootstrap.group(bossGroup, workerGroup)*

*.channel(classOf[NioServerSocketChannel]) //创建NioServerSocketChannel*

*.childHandler(new ChannelInitializer[SocketChannel] { //绑定I/O事件处理类*

*override def initChannel(sc: SocketChannel): Unit ={*

*sc.pipeline().addLast(*

*new ServerHandler*

*)*

*}*

*})*

*//绑定端口，调用sync方法等待绑定操作完成*

*val channelFuture = bootstrap.bind(host,port).sync()*

*channelFuture.channel().closeFuture().sync() //等待服务关闭*

*} finally {*

*println("Netty Server stoped")*

*bossGroup.shutdownGracefully()*

*workerGroup.shutdownGracefully()*

*}*

*}*

*}*

*object NettyServer {*

*def main(args: Array[String]): Unit = {*

*val host = "127.0.0.1"*

*val port = "6532".toInt*

*val server = new NettyServer*

*server.bind(host,port)*

*}*

*}*

## 2.3 Client Handler

*class ClientHandler extends ChannelInboundHandlerAdapter {*

*override def channelActive(ctx: ChannelHandlerContext): Unit = {*

*println("Channel Active")*

*val content = "Hello Server"*

*ctx.writeAndFlush(Unpooled.copiedBuffer(content.getBytes("UTF-8")))*

*//发送字符串，先封装成ByteBf*

*}*

*override def channelRead(ctx: ChannelHandlerContext, msg: Any): Unit = {*

*println("Channel Read")*

*val byteBuf = msg.asInstanceOf[ByteBuf]*

*val bytes = new Array[Byte](byteBuf.readableBytes())*

*byteBuf.readBytes(bytes)*

*val message = new String(bytes, "UTF-8")*

*println(message)*

*}*

*override def exceptionCaught(ctx: ChannelHandlerContext, cause: Throwable): Unit = {*

*println("Exception caught")*

*ctx.close()*

*}*

*}*

## 2.4 Netty Client

*class NettyClient {*

*def connect(host: String, port: Int): Unit = {*

*//创建客户端NIO线程组*

*val eventGroup = new NioEventLoopGroup*

*//创建客户端辅助启动类*

*val bootStrap = new Bootstrap*

*try {*

*//将NIO线程组传入到BootStrap中*

*bootStrap.group(eventGroup)*

*.channel(classOf[NioSocketChannel]) //创建NioSocketChannel*

*.handler(new ChannelInitializer[SocketChannel] {*

*override def initChannel(sc: SocketChannel): Unit = {*

*sc.pipeline().addLast( //绑定I/O事件处理类*

*new ClientHandler*

*)*

*}*

*})*

*//发起异步连接操作*

*val channelFuture = bootStrap.connect(host,port).sync()*

*channelFuture.channel().closeFuture().sync()*

*} finally{*

*eventGroup.shutdownGracefully()*

*}*

*}*

*}*

*object NettyClient {*

*def main(args:Array[String]): Unit = {*

*val host = "localhost"*

*val port = "6532".toInt*

*val client = new NettyClient*

*client.connect(host,port)*

*}*

*}*

http://ifeve.com/netty-2-3-1/

http://blog.csdn.net/qq\_21383435/article/details/78995552

http://netty.io/wiki/user-guide-for-4.x.html