**一、什么是Netty**

Netty是由[JBOSS](https://baike.baidu.com/item/JBOSS" \t "https://baike.baidu.com/item/Netty/_blank)提供的一个[java开源](https://baike.baidu.com/item/java%E5%BC%80%E6%BA%90" \t "https://baike.baidu.com/item/Netty/_blank)框架。Netty提供异步的、[事件驱动](https://baike.baidu.com/item/%E4%BA%8B%E4%BB%B6%E9%A9%B1%E5%8A%A8" \t "https://baike.baidu.com/item/Netty/_blank)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](https://baike.baidu.com/item/%E7%BD%91%E7%BB%9C%E6%9C%8D%E5%8A%A1%E5%99%A8" \t "https://baike.baidu.com/item/Netty/_blank)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 可以确保你快速和简单的开发出一个网络应用，例如实现了某种协议的客户，[服务端](https://baike.baidu.com/item/%E6%9C%8D%E5%8A%A1%E7%AB%AF" \t "https://baike.baidu.com/item/Netty/_blank)应用。Netty相当简化和流线化了网络应用的编程开发过程，例如，TCP和UDP的socket服务开发。

“快速”和“简单”并不用产生维护性或性能上的问题。Netty 是一个吸收了多种协议的实现经验，这些协议包括FTP,SMTP,HTTP，各种二进制，文本协议，并经过相当精心设计的项目，最终，Netty 成功的找到了一种方式，在保证易于开发的同时还保证了其应用的性能，稳定性和伸缩性。

1. **使用Netty**

1、下载Netty的jar包，下载地址：<http://netty.io/>

2、Netty实现通信的步骤

(1)创建两个NIO事件组，一个专门用于网络事件处理（接受客户端的连接），另一个则进行网络通信读写。

(2)创建一个ServerBootstrap对象，配置Netty的一系列参数，例如接受传出数据的缓存大小等等。

(3)创建一个实际处理数据的类ChannelInitializer，进行初始化的准备工作，比如设置接受传出数据的字符集、格式、实际处理数据的接口

(4)绑定端口，执行同步阻塞方法等待服务器端启动即可。

三、Helloworld

1、服务器端

(1)Server类

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 Server {

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

//1 创建两个Nio事件组

//一个是用于处理服务器端接收客户端连接的

//一个是进行网络通信的（网络读写的）

EventLoopGroup pGroup = new NioEventLoopGroup();

EventLoopGroup cGroup = new NioEventLoopGroup();

//2 创建辅助工具类，用于服务器通道的一系列配置

ServerBootstrap b = new ServerBootstrap();

b.group(pGroup, cGroup) //绑定俩个事件组

.channel(NioServerSocketChannel.class) //指定NIO的模式，可以监听新进来的TCP连接的通道

.option(ChannelOption.SO\_BACKLOG, 1024) //设置tcp缓冲区，将每个连接的客户端对象放到TCP缓冲区中，单位byte

.option(ChannelOption.SO\_SNDBUF, 32\*1024) //设置发送数据缓冲大小 .option(ChannelOption.SO\_RCVBUF, 32\*1024) //这是接收数据缓冲大小

.option(ChannelOption.SO\_KEEPALIVE, true) //保持连接

.childHandler(new ChannelInitializer<SocketChannel>() {//用来监听已经连接的客户端的动作和状态。

@Override

protected void initChannel(SocketChannel sc) throws Exception {

//3 在这里配置具体数据接收方法的处理

sc.pipeline().addLast(new ServerHandler());

}

});

//4 进行绑定，绑定端口8765

ChannelFuture cf1 = b.bind(8765).sync();

//5 等待关闭

cf1.channel().closeFuture().sync();

//6 释放

pGroup.shutdownGracefully();

cGroup.shutdownGracefully();

}

}

1. ServerHandler类：服务器端处理器类

import io.netty.buffer.ByteBuf;

import io.netty.buffer.Unpooled;

import io.netty.channel.ChannelFutureListener;

import io.netty.channel.ChannelHandlerAdapter;

import io.netty.channel.ChannelHandlerContext;

public class ServerHandler extends ChannelHandlerAdapter {

//socket建立连接时执行

@Override

public void channelActive(ChannelHandlerContext ctx) throws Exception {

System.out.println("server channel active... ");

}

//从当前Channel的对端读取消息。

@Override

public void channelRead(ChannelHandlerContext ctx, Object msg)

throws Exception {

ByteBuf buf = (ByteBuf) msg;

byte[] req = new byte[buf.readableBytes()];

//将buf读到req字节数组中

buf.readBytes(req);

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

System.out.println("Server :" + body );

//服务器端给客户端的响应

String response = "Hi client!" ;

ctx.writeAndFlush(Unpooled.copiedBuffer(response.getBytes()));

}

//消息读取完毕后执行。

@Override

public void channelReadComplete(ChannelHandlerContext ctx)

throws Exception {

System.out.println("读完了");

ctx.flush();

}

//处理异常.

@Override

public void exceptionCaught(ChannelHandlerContext ctx, Throwable t)

throws Exception {

ctx.close();

}

}

1. 客户端

(1)Client类

import io.netty.bootstrap.Bootstrap;

import io.netty.buffer.Unpooled;

import io.netty.channel.ChannelFuture;

import io.netty.channel.ChannelInitializer;

import io.netty.channel.EventLoopGroup;

import io.netty.channel.nio.NioEventLoopGroup;

import io.netty.channel.socket.SocketChannel;

import io.netty.channel.socket.nio.NioSocketChannel;

public class Client {

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

//创建一个事件组

EventLoopGroup group = new NioEventLoopGroup();

Bootstrap b = new Bootstrap();

b.group(group)

.channel(NioSocketChannel.class) //指定NIO的模式

.handler(new ChannelInitializer<SocketChannel>() {//用来监听已经连接的服务端的动作和状态。

@Override

protected void initChannel(SocketChannel sc) throws Exception {

sc.pipeline().addLast(new ClientHandler());

}

});

//发起连接，指定连接服务器端的端口号

ChannelFuture cf1 = b.connect("127.0.0.1", 8765).sync();

//发送消息

cf1.channel().write(Unpooled.copiedBuffer("hello netty!!".getBytes()));

cf1.channel().flush();

cf1.channel().closeFuture().sync();

//6 释放

group.shutdownGracefully();

}

}

1. ClientHandler：客户端处理器类

import io.netty.buffer.ByteBuf;

import io.netty.buffer.Unpooled;

import io.netty.channel.ChannelHandlerAdapter;

import io.netty.channel.ChannelHandlerContext;

import io.netty.channel.ChannelPromise;

import io.netty.util.ReferenceCountUtil;

public class ClientHandler extends ChannelHandlerAdapter{

//通道激活时触发，当客户端connect成功后，服务端就会接收到这个事件

@Override

public void channelActive(ChannelHandlerContext ctx) throws Exception {

}

//当收到对方发来的数据后，就会触发，参数msg就是发来的信息

@Override

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

ByteBuf buf = (ByteBuf) msg;

byte[] req = new byte[buf.readableBytes()];

buf.readBytes(req);

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

System.out.println("Client :" + body );

}

//channelRead执行后触发

@Override

public void channelReadComplete(ChannelHandlerContext ctx) throws Exception {

}

//抛出异常时触发，做一些错误处理

@Override

public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)

throws Exception {

ctx.close();

}

}

1. 写三次只刷新一次
2. 客户端

修改Client类

import io.netty.bootstrap.Bootstrap;

import io.netty.buffer.Unpooled;

import io.netty.channel.ChannelFuture;

import io.netty.channel.ChannelInitializer;

import io.netty.channel.EventLoopGroup;

import io.netty.channel.nio.NioEventLoopGroup;

import io.netty.channel.socket.SocketChannel;

import io.netty.channel.socket.nio.NioSocketChannel;

public class Client {

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

//创建一个事件组

EventLoopGroup group = new NioEventLoopGroup();

Bootstrap b = new Bootstrap();

b.group(group)

.channel(NioSocketChannel.class)

.handler(new ChannelInitializer<SocketChannel>() {

@Override

protected void initChannel(SocketChannel sc) throws Exception {

sc.pipeline().addLast(new ClientHandler());

}

});

//发起连接

ChannelFuture cf1 = b.connect("127.0.0.1", 8765).sync();

//发送消息

//写三次只刷新一次

cf1.channel().write(Unpooled.copiedBuffer("hello netty!!".getBytes()));

cf1.channel().write(Unpooled.copiedBuffer("hello netty!!".getBytes()));

cf1.channel().write(Unpooled.copiedBuffer("hello netty!!".getBytes()));

cf1.channel().flush();

cf1.channel().closeFuture().sync();

group.shutdownGracefully();

}

}

执行结果：



1. 客户端的应用程序主动关闭

修改ServerHandler

import io.netty.buffer.ByteBuf;

import io.netty.buffer.Unpooled;

import io.netty.channel.ChannelFutureListener;

import io.netty.channel.ChannelHandlerAdapter;

import io.netty.channel.ChannelHandlerContext;

public class ServerHandler extends ChannelHandlerAdapter {

@Override

public void channelActive(ChannelHandlerContext ctx) throws Exception {

System.out.println("server channel active... ");

}

@Override

public void channelRead(ChannelHandlerContext ctx, Object msg)

throws Exception {

ByteBuf buf = (ByteBuf) msg;

byte[] req = new byte[buf.readableBytes()];

buf.readBytes(req);

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

System.out.println("Server :" + body );

String response = "Hi client!" ;

//.addListener(ChannelFutureListener.CLOSE);客户端执行结束后客户端程序结束

ctx.writeAndFlush(Unpooled.copiedBuffer(response.getBytes())).addListener(ChannelFutureListener.CLOSE);

}

@Override

public void channelReadComplete(ChannelHandlerContext ctx)

throws Exception {

System.out.println("读完了");

ctx.flush();

}

@Override

public void exceptionCaught(ChannelHandlerContext ctx, Throwable t)

throws Exception {

ctx.close();

}

}

1. 服务器端绑定两个端口

(1)修改Server类：绑定第二个端口

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 Server {

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

//1 创建线两个事件组

//一个是用于处理服务器端接收客户端连接的

//一个是进行网络通信的（网络读写的）

EventLoopGroup pGroup = new NioEventLoopGroup();

EventLoopGroup cGroup = new NioEventLoopGroup();

//2 创建辅助工具类，用于服务器通道的一系列配置

ServerBootstrap b = new ServerBootstrap();

b.group(pGroup, cGroup) //绑定俩个事件组

.channel(NioServerSocketChannel.class) //指定NIO的模式

.option(ChannelOption.SO\_BACKLOG, 1024) //设置tcp缓冲区

.option(ChannelOption.SO\_SNDBUF, 32\*1024) //设置发送缓冲大小

.option(ChannelOption.SO\_RCVBUF, 32\*1024) //这是接收缓冲大小

.option(ChannelOption.SO\_KEEPALIVE, true) //保持连接

.childHandler(new ChannelInitializer<SocketChannel>() {

@Override

protected void initChannel(SocketChannel sc) throws Exception {

//3 在这里配置具体数据接收方法的处理

sc.pipeline().addLast(new ServerHandler());

}

});

//4 进行绑定

ChannelFuture cf1 = b.bind(8765).sync();

ChannelFuture cf2 = b.bind(8764).sync();//开第二个端口

//5 等待关闭

cf1.channel().closeFuture().sync();

cf2.channel().closeFuture().sync();

//释放

pGroup.shutdownGracefully();

cGroup.shutdownGracefully();

}

}

1. 修改Client类：向第二个服务器端口发送消息

import io.netty.bootstrap.Bootstrap;

import io.netty.buffer.Unpooled;

import io.netty.channel.ChannelFuture;

import io.netty.channel.ChannelInitializer;

import io.netty.channel.EventLoopGroup;

import io.netty.channel.nio.NioEventLoopGroup;

import io.netty.channel.socket.SocketChannel;

import io.netty.channel.socket.nio.NioSocketChannel;

public class Client {

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

//创建一个事件组

EventLoopGroup group = new NioEventLoopGroup();

Bootstrap b = new Bootstrap();

b.group(group)

.channel(NioSocketChannel.class)

.handler(new ChannelInitializer<SocketChannel>() {

@Override

protected void initChannel(SocketChannel sc) throws Exception {

sc.pipeline().addLast(new ClientHandler());

}

});

//发起连接

ChannelFuture cf1 = b.connect("127.0.0.1", 8765).sync();

//连接第二个服务端口

ChannelFuture cf2 = b.connect("127.0.0.1", 8764).sync();

//发送消息

cf1.channel().writeAndFlush(Unpooled.copiedBuffer("hello netty!!".getBytes()));

//向第二个服务端口发送消息

cf2.channel().writeAndFlush(Unpooled.copiedBuffer("hello world!!".getBytes()));

cf1.channel().closeFuture().sync();

cf2.channel().closeFuture().sync();

group.shutdownGracefully();

}

}

1. TCP粘包、拆包

在业务上，一个完整的包可能会被TCP分成多个包进行发送，也可能把多个小包封装成一个大的数据包发送出去，这就是TCP粘包、拆包。

1. 通过Thread.sleep()分隔发送消息（不推荐）

修改Client类

import io.netty.bootstrap.Bootstrap;

import io.netty.buffer.Unpooled;

import io.netty.channel.ChannelFuture;

import io.netty.channel.ChannelInitializer;

import io.netty.channel.EventLoopGroup;

import io.netty.channel.nio.NioEventLoopGroup;

import io.netty.channel.socket.SocketChannel;

import io.netty.channel.socket.nio.NioSocketChannel;

public class Client {

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

//创建一个事件组

EventLoopGroup group = new NioEventLoopGroup();

Bootstrap b = new Bootstrap();

b.group(group)

.channel(NioSocketChannel.class)

.handler(new ChannelInitializer<SocketChannel>() {

@Override

protected void initChannel(SocketChannel sc) throws Exception {

sc.pipeline().addLast(new ClientHandler());

}

});

//发起连接

ChannelFuture cf1 = b.connect("127.0.0.1", 8765).sync();

//服务器端显示连续字符串：hello netty!!hello netty!!hello netty!!

/\*cf1.channel().write(Unpooled.copiedBuffer("hello netty!!".getBytes()));

cf1.channel().write(Unpooled.copiedBuffer("hello netty!!".getBytes()));

cf1.channel().write(Unpooled.copiedBuffer("hello netty!!".getBytes()));\*/

//发送消息

cf1.channel().writeAndFlush(Unpooled.copiedBuffer("hello netty!!".getBytes()));

//分隔发送消息

Thread.sleep(1000);

cf1.channel().writeAndFlush(Unpooled.copiedBuffer("777".getBytes()));

Thread.sleep(2000);

cf1.channel().writeAndFlush(Unpooled.copiedBuffer("888".getBytes()));

cf1.channel().closeFuture().sync();

group.shutdownGracefully();

}

}

1. 粘包拆包的解决方案一：在包尾部增加特殊字符进行分割，使用DelimiterBasedFrameDecoder类

(1)Server类

import java.nio.ByteBuffer;

import io.netty.bootstrap.ServerBootstrap;

import io.netty.buffer.ByteBuf;

import io.netty.buffer.Unpooled;

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;

import io.netty.handler.codec.DelimiterBasedFrameDecoder;

import io.netty.handler.codec.FixedLengthFrameDecoder;

import io.netty.handler.codec.string.StringDecoder;

import io.netty.handler.codec.string.StringEncoder;

public class Server {

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

//1 创建2个事件组，一个是负责接收客户端的连接。一个是负责进行数据传输的

EventLoopGroup pGroup = new NioEventLoopGroup();

EventLoopGroup cGroup = new NioEventLoopGroup();

//2 创建服务器辅助类

ServerBootstrap b = new ServerBootstrap();

b.group(pGroup, cGroup)

.channel(NioServerSocketChannel.class)

.option(ChannelOption.SO\_BACKLOG, 1024)

.option(ChannelOption.SO\_SNDBUF, 32\*1024)

.option(ChannelOption.SO\_RCVBUF, 32\*1024)

.childHandler(new ChannelInitializer<SocketChannel>() {

@Override

protected void initChannel(SocketChannel sc) throws Exception {

//设置特殊分隔符

ByteBuf buf = Unpooled.copiedBuffer("$\_".getBytes());

//设置缓冲大小

sc.pipeline().addLast(new DelimiterBasedFrameDecoder(1024, buf));

//设置字符串形式的解码

sc.pipeline().addLast(new StringDecoder());

sc.pipeline().addLast(new ServerHandler());

}

});

//4 绑定连接

ChannelFuture cf = b.bind(8765).sync();

//等待服务器监听端口关闭

cf.channel().closeFuture().sync();

pGroup.shutdownGracefully();

cGroup.shutdownGracefully();

}

}

1. ServerHandler类

import io.netty.buffer.Unpooled;

import io.netty.channel.ChannelHandlerAdapter;

import io.netty.channel.ChannelHandlerContext;

public class ServerHandler extends ChannelHandlerAdapter {

@Override

public void channelActive(ChannelHandlerContext ctx) throws Exception {

System.out.println(" server channel active... ");

}

@Override

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

String request = (String)msg; //解析的是String类型

System.out.println("Server :" + request);

String response = "我是响应数据$\_";

ctx.writeAndFlush(Unpooled.copiedBuffer(response.getBytes()));

}

@Override

public void channelReadComplete(ChannelHandlerContext ctx) throws Exception {

}

@Override

public void exceptionCaught(ChannelHandlerContext ctx, Throwable t) throws Exception {

ctx.close();

}

}

1. Client类

import io.netty.bootstrap.Bootstrap;

import io.netty.buffer.ByteBuf;

import io.netty.buffer.Unpooled;

import io.netty.channel.ChannelFuture;

import io.netty.channel.ChannelInitializer;

import io.netty.channel.EventLoopGroup;

import io.netty.channel.nio.NioEventLoopGroup;

import io.netty.channel.socket.SocketChannel;

import io.netty.channel.socket.nio.NioSocketChannel;

import io.netty.handler.codec.DelimiterBasedFrameDecoder;

import io.netty.handler.codec.FixedLengthFrameDecoder;

import io.netty.handler.codec.string.StringDecoder;

import io.netty.handler.codec.string.StringEncoder;

public class Client {

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

EventLoopGroup group = new NioEventLoopGroup();

Bootstrap b = new Bootstrap();

b.group(group)

.channel(NioSocketChannel.class)

.handler(new ChannelInitializer<SocketChannel>() {

@Override

protected void initChannel(SocketChannel sc) throws Exception {

//分隔符

ByteBuf buf = Unpooled.copiedBuffer("$\_".getBytes());

sc.pipeline().addLast(new DelimiterBasedFrameDecoder(1024, buf));

sc.pipeline().addLast(new StringDecoder());

sc.pipeline().addLast(new ClientHandler());

}

});

ChannelFuture cf = b.connect("127.0.0.1", 8765).sync();

cf.channel().writeAndFlush(Unpooled.wrappedBuffer("aaa$\_".getBytes()));

cf.channel().writeAndFlush(Unpooled.wrappedBuffer("bbbb$\_".getBytes()));

cf.channel().writeAndFlush(Unpooled.wrappedBuffer("cccc$\_".getBytes()));

//等待客户端端口关闭

cf.channel().closeFuture().sync();

group.shutdownGracefully();

}

}

(4)ClientHandler类

import io.netty.channel.ChannelHandlerAdapter;

import io.netty.channel.ChannelHandlerContext;

import io.netty.util.ReferenceCountUtil;

public class ClientHandler extends ChannelHandlerAdapter{

@Override

public void channelActive(ChannelHandlerContext ctx) throws Exception {

System.out.println("client channel active... ");

}

@Override

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

String response = (String)msg;

System.out.println("Client: " + response);

}

@Override

public void channelReadComplete(ChannelHandlerContext ctx) throws Exception {

}

@Override

public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws Exception {

ctx.close();

}

}

1. 粘包拆包的解决方案二：消息定长，例如每个报文的大小固定为200个字节，如果不够，空位补空格，使用FixedLengthFrameDecoder类
2. Server类

import java.nio.ByteBuffer;

import io.netty.bootstrap.ServerBootstrap;

import io.netty.buffer.ByteBuf;

import io.netty.buffer.Unpooled;

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;

import io.netty.handler.codec.DelimiterBasedFrameDecoder;

import io.netty.handler.codec.FixedLengthFrameDecoder;

import io.netty.handler.codec.string.StringDecoder;

import io.netty.handler.codec.string.StringEncoder;

public class Server {

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

//1 创建2个事件组，一个是负责接收客户端的连接。一个是负责进行数据传输的

EventLoopGroup pGroup = new NioEventLoopGroup();

EventLoopGroup cGroup = new NioEventLoopGroup();

//2 创建服务器辅助类

ServerBootstrap b = new ServerBootstrap();

b.group(pGroup, cGroup)

.channel(NioServerSocketChannel.class)

.option(ChannelOption.SO\_BACKLOG, 1024)

.option(ChannelOption.SO\_SNDBUF, 32\*1024)

.option(ChannelOption.SO\_RCVBUF, 32\*1024)

.childHandler(new ChannelInitializer<SocketChannel>() {

@Override

protected void initChannel(SocketChannel sc) throws Exception {

//设置定长字符串接收

sc.pipeline().addLast(new FixedLengthFrameDecoder(5));

//设置字符串形式的解码

sc.pipeline().addLast(new StringDecoder());

sc.pipeline().addLast(new ServerHandler());

}

});

//4 绑定连接

ChannelFuture cf = b.bind(8765).sync();

//等待服务器监听端口关闭

cf.channel().closeFuture().sync();

pGroup.shutdownGracefully();

cGroup.shutdownGracefully();

}

}

1. ServerHandler：

import io.netty.buffer.Unpooled;

import io.netty.channel.ChannelHandlerAdapter;

import io.netty.channel.ChannelHandlerContext;

public class ServerHandler extends ChannelHandlerAdapter {

@Override

public void channelActive(ChannelHandlerContext ctx) throws Exception {

System.out.println(" server channel active... ");

}

@Override

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

String request = (String)msg;

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

String response = request ;

ctx.writeAndFlush(Unpooled.copiedBuffer(response.getBytes()));

}

@Override

public void channelReadComplete(ChannelHandlerContext ctx) throws Exception {

}

@Override

public void exceptionCaught(ChannelHandlerContext ctx, Throwable t) throws Exception {

}

}

1. Client类

import io.netty.bootstrap.Bootstrap;

import io.netty.buffer.ByteBuf;

import io.netty.buffer.Unpooled;

import io.netty.channel.ChannelFuture;

import io.netty.channel.ChannelInitializer;

import io.netty.channel.EventLoopGroup;

import io.netty.channel.nio.NioEventLoopGroup;

import io.netty.channel.socket.SocketChannel;

import io.netty.channel.socket.nio.NioSocketChannel;

import io.netty.handler.codec.DelimiterBasedFrameDecoder;

import io.netty.handler.codec.FixedLengthFrameDecoder;

import io.netty.handler.codec.string.StringDecoder;

import io.netty.handler.codec.string.StringEncoder;

public class Client {

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

EventLoopGroup group = new NioEventLoopGroup();

Bootstrap b = new Bootstrap();

b.group(group)

.channel(NioSocketChannel.class)

.handler(new ChannelInitializer<SocketChannel>() {

@Override

protected void initChannel(SocketChannel sc) throws Exception {

sc.pipeline().addLast(new FixedLengthFrameDecoder(5));

sc.pipeline().addLast(new StringDecoder());

sc.pipeline().addLast(new ClientHandler());

}

});

ChannelFuture cf = b.connect("127.0.0.1", 8765).sync();

cf.channel().writeAndFlush(Unpooled.wrappedBuffer("aaaaabbbbb".getBytes()));

//如果发送“ccccccc”只能收到5个c，可以使用空格补位

cf.channel().writeAndFlush(Unpooled.copiedBuffer("ccccccc ".getBytes()));

//等待客户端端口关闭

cf.channel().closeFuture().sync();

group.shutdownGracefully();

}

}

1. ClientHandler类：与方案一相同

import io.netty.channel.ChannelHandlerAdapter;

import io.netty.channel.ChannelHandlerContext;

public class ClientHandler extends ChannelHandlerAdapter{

@Override

public void channelActive(ChannelHandlerContext ctx) throws Exception {

System.out.println("client channel active... ");

}

@Override

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

String response = (String)msg;

System.out.println("Client: " + response);

}

@Override

public void channelReadComplete(ChannelHandlerContext ctx) throws Exception {

}

@Override

public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws Exception {

}

}

1. Netty编解码技术

编解码技术：就是Java序列化技术，序列化目的就两个，第一进行网络传输，第二对象持久化。

编解码框架：JBoss的Marshalling

十一、序列化：客户端将对象传递为服务器端

(1)Req类：客户端发送给服务器端的对象

import java.io.Serializable;

//必须实现Serializable接口

public class Req implements Serializable{

private static final long SerialVersionUID = 1L;

private String id ;

private String name ;

private String requestMessage ;

private byte[] attachment;

public String getId() {

return id;

}

public void setId(String id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getRequestMessage() {

return requestMessage;

}

public void setRequestMessage(String requestMessage) {

this.requestMessage = requestMessage;

}

public byte[] getAttachment() {

return attachment;

}

public void setAttachment(byte[] attachment) {

this.attachment = attachment;

}

}

1. Resp类：服务器端返回给客户端的对象

import java.io.Serializable;

//必须实现Serializable接口

public class Resp implements Serializable{

private static final long serialVersionUID = 1L;

private String id;

private String name;

private String responseMessage;

public String getId() {

return id;

}

public void setId(String id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getResponseMessage() {

return responseMessage;

}

public void setResponseMessage(String responseMessage) {

this.responseMessage = responseMessage;

}

}

1. Server类

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;

import io.netty.handler.logging.LogLevel;

import io.netty.handler.logging.LoggingHandler;

public class Server {

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

EventLoopGroup pGroup = new NioEventLoopGroup();

EventLoopGroup cGroup = new NioEventLoopGroup();

ServerBootstrap b = new ServerBootstrap();

b.group(pGroup, cGroup)

.channel(NioServerSocketChannel.class)

.option(ChannelOption.SO\_BACKLOG, 1024)

//设置打印日志

.handler(new LoggingHandler(LogLevel.INFO))

.childHandler(new ChannelInitializer<SocketChannel>() {

protected void initChannel(SocketChannel sc) throws Exception {

//设置序列化框架

sc.pipeline().addLast(MarshallingCodeCFactory.buildMarshallingDecoder());

sc.pipeline().addLast(MarshallingCodeCFactory.buildMarshallingEncoder());

sc.pipeline().addLast(new ServerHandler());

}

});

ChannelFuture cf = b.bind(8765).sync();

cf.channel().closeFuture().sync();

pGroup.shutdownGracefully();

cGroup.shutdownGracefully();

}

}

1. ServerHandler类

import io.netty.channel.ChannelHandlerAdapter;

import io.netty.channel.ChannelHandlerContext;

import java.io.File;

import java.io.FileOutputStream;

import bhz.utils.GzipUtils;

public class ServerHandler extends ChannelHandlerAdapter{

@Override

public void channelActive(ChannelHandlerContext ctx) throws Exception {

}

@Override

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

Req req = (Req)msg;

System.out.println("Server : " + req.getId() + ", " + req.getName() + ", " + req.getRequestMessage());

Resp resp = new Resp();

resp.setId(req.getId());

resp.setName("resp" + req.getId());

resp.setResponseMessage("响应内容" + req.getId());

ctx.writeAndFlush(resp);

}

@Override

public void channelReadComplete(ChannelHandlerContext ctx) throws Exception {

}

@Override

public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws Exception {

ctx.close();

}

}

1. Client类

import io.netty.bootstrap.Bootstrap;

import io.netty.channel.ChannelFuture;

import io.netty.channel.ChannelInitializer;

import io.netty.channel.EventLoopGroup;

import io.netty.channel.nio.NioEventLoopGroup;

import io.netty.channel.socket.SocketChannel;

import io.netty.channel.socket.nio.NioSocketChannel;

import java.io.File;

import java.io.FileInputStream;

import bhz.utils.GzipUtils;

public class Client {

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

EventLoopGroup group = new NioEventLoopGroup();

Bootstrap b = new Bootstrap();

b.group(group)

.channel(NioSocketChannel.class)

.handler(new ChannelInitializer<SocketChannel>() {

@Override

protected void initChannel(SocketChannel sc) throws Exception {

//设置序列化框架

sc.pipeline().addLast(MarshallingCodeCFactory.buildMarshallingDecoder());

sc.pipeline().addLast(MarshallingCodeCFactory.buildMarshallingEncoder());

sc.pipeline().addLast(new ClientHandler());

}

});

ChannelFuture cf = b.connect("127.0.0.1", 8765).sync();

//将5个对象冲刷到服务器端

for(int i = 0; i < 5; i++ ){

Req req = new Req();

req.setId("" + i);

req.setName("pro" + i);

req.setRequestMessage("数据信息" + i);

cf.channel().writeAndFlush(req);

}

cf.channel().closeFuture().sync();

group.shutdownGracefully();

}

}

1. ClientHandler类

import io.netty.channel.ChannelHandlerAdapter;

import io.netty.channel.ChannelHandlerContext;

import io.netty.util.ReferenceCountUtil;

public class ClientHandler extends ChannelHandlerAdapter{

@Override

public void channelActive(ChannelHandlerContext ctx) throws Exception {

}

@Override

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

Resp resp = (Resp)msg;

System.out.println("Client : " + resp.getId() + ", " + resp.getName() + ", " + resp.getResponseMessage());

}

@Override

public void channelReadComplete(ChannelHandlerContext ctx) throws Exception {

}

@Override

public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws Exception {

ctx.close();

}

}

1. MarshallingCodeCFactory类：Marshalling工厂

import io.netty.handler.codec.marshalling.DefaultMarshallerProvider;

import io.netty.handler.codec.marshalling.DefaultUnmarshallerProvider;

import io.netty.handler.codec.marshalling.MarshallerProvider;

import io.netty.handler.codec.marshalling.MarshallingDecoder;

import io.netty.handler.codec.marshalling.MarshallingEncoder;

import io.netty.handler.codec.marshalling.UnmarshallerProvider;

import org.jboss.marshalling.MarshallerFactory;

import org.jboss.marshalling.Marshalling;

import org.jboss.marshalling.MarshallingConfiguration;

public final class MarshallingCodeCFactory {

/\*\*

\* 创建Jboss Marshalling解码器MarshallingDecoder

\* @return MarshallingDecoder

\*/

public static MarshallingDecoder buildMarshallingDecoder() {

//首先通过Marshalling工具类的方法获取Marshalling实例对象 参数serial标识创建的是java序列化工厂对象。

final MarshallerFactory marshallerFactory = Marshalling.getProvidedMarshallerFactory("serial");

//创建了MarshallingConfiguration对象，配置了版本号为5

final MarshallingConfiguration configuration = new MarshallingConfiguration();

configuration.setVersion(5);

//根据marshallerFactory和configuration创建provider

UnmarshallerProvider provider = new DefaultUnmarshallerProvider(marshallerFactory, configuration);

//构建Netty的MarshallingDecoder对象，俩个参数分别为provider和单个消息序列化后的最大长度

MarshallingDecoder decoder = new MarshallingDecoder(provider, 1024 \* 1024 \* 1);

return decoder;

}

/\*\*

\* 创建Jboss Marshalling编码器MarshallingEncoder

\* @return MarshallingEncoder

\*/

public static MarshallingEncoder buildMarshallingEncoder() {

final MarshallerFactory marshallerFactory = Marshalling.getProvidedMarshallerFactory("serial");

final MarshallingConfiguration configuration = new MarshallingConfiguration();

configuration.setVersion(5);

MarshallerProvider provider = new DefaultMarshallerProvider(marshallerFactory, configuration);

//构建Netty的MarshallingEncoder对象，MarshallingEncoder用于实现序列化接口的POJO对象序列化为二进制数组

MarshallingEncoder encoder = new MarshallingEncoder(provider);

return encoder;

}

}

1. 序列化发送附件(压缩、解压文件)

(1)序列化的对象：Req和Reps两个类

(2)GzipUtils：压缩和解压的工具类

import java.io.ByteArrayInputStream;

import java.io.ByteArrayOutputStream;

import java.io.File;

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.util.zip.GZIPInputStream;

import java.util.zip.GZIPOutputStream;

public class GzipUtils {

//压缩文件

public static byte[] gzip(byte[] data) throws Exception{

ByteArrayOutputStream bos = new ByteArrayOutputStream();

GZIPOutputStream gzip = new GZIPOutputStream(bos);

gzip.write(data);

gzip.finish();

gzip.close();

byte[] ret = bos.toByteArray();

bos.close();

return ret;

}

//解压文件

public static byte[] ungzip(byte[] data) throws Exception{

ByteArrayInputStream bis = new ByteArrayInputStream(data);

GZIPInputStream gzip = new GZIPInputStream(bis);

byte[] buf = new byte[1024];

int num = -1;

ByteArrayOutputStream bos = new ByteArrayOutputStream();

while((num = gzip.read(buf, 0 , buf.length)) != -1 ){

bos.write(buf, 0, num);

}

gzip.close();

bis.close();

byte[] ret = bos.toByteArray();

bos.flush();

bos.close();

return ret;

}

}

1. Server类：与序列化无变化

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;

import io.netty.handler.logging.LogLevel;

import io.netty.handler.logging.LoggingHandler;

public class Server {

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

EventLoopGroup pGroup = new NioEventLoopGroup();

EventLoopGroup cGroup = new NioEventLoopGroup();

ServerBootstrap b = new ServerBootstrap();

b.group(pGroup, cGroup)

.channel(NioServerSocketChannel.class)

.option(ChannelOption.SO\_BACKLOG, 1024)

//设置日志

.handler(new LoggingHandler(LogLevel.INFO))

.childHandler(new ChannelInitializer<SocketChannel>() {

protected void initChannel(SocketChannel sc) throws Exception {

//设置序列化框架

sc.pipeline().addLast(MarshallingCodeCFactory.buildMarshallingDecoder());

sc.pipeline().addLast(MarshallingCodeCFactory.buildMarshallingEncoder());

sc.pipeline().addLast(new ServerHandler());

}

});

ChannelFuture cf = b.bind(8765).sync();

cf.channel().closeFuture().sync();

pGroup.shutdownGracefully();

cGroup.shutdownGracefully();

}

}

1. ServerHandler类

import io.netty.channel.ChannelHandlerAdapter;

import io.netty.channel.ChannelHandlerContext;

import java.io.File;

import java.io.FileOutputStream;

import bhz.utils.GzipUtils;

public class ServerHandler extends ChannelHandlerAdapter{

@Override

public void channelActive(ChannelHandlerContext ctx) throws Exception {

}

@Override

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

Req req = (Req)msg;

System.out.println("Server : " + req.getId() + ", " + req.getName() + ", " + req.getRequestMessage());

//解压

byte[] attachment = GzipUtils.ungzip(req.getAttachment());

//将解压文件写到指定的path

String path = System.getProperty("user.dir") + File.separatorChar + "receive" + File.separatorChar + "001.jpg";

FileOutputStream fos = new FileOutputStream(path);

fos.write(attachment);

fos.close();

Resp resp = new Resp();

resp.setId(req.getId());

resp.setName("resp" + req.getId());

resp.setResponseMessage("响应内容" + req.getId());

ctx.writeAndFlush(resp);//.addListener(ChannelFutureListener.CLOSE);

}

@Override

public void channelReadComplete(ChannelHandlerContext ctx) throws Exception {

}

@Override

public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws Exception {

ctx.close();

}

}

1. Client类

import io.netty.bootstrap.Bootstrap;

import io.netty.channel.ChannelFuture;

import io.netty.channel.ChannelInitializer;

import io.netty.channel.EventLoopGroup;

import io.netty.channel.nio.NioEventLoopGroup;

import io.netty.channel.socket.SocketChannel;

import io.netty.channel.socket.nio.NioSocketChannel;

import java.io.File;

import java.io.FileInputStream;

import bhz.utils.GzipUtils;

public class Client {

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

EventLoopGroup group = new NioEventLoopGroup();

Bootstrap b = new Bootstrap();

b.group(group)

.channel(NioSocketChannel.class)

.handler(new ChannelInitializer<SocketChannel>() {

@Override

protected void initChannel(SocketChannel sc) throws Exception {

//设置序列化框架

sc.pipeline().addLast(MarshallingCodeCFactory.buildMarshallingDecoder());

sc.pipeline().addLast(MarshallingCodeCFactory.buildMarshallingEncoder());

sc.pipeline().addLast(new ClientHandler());

}

});

ChannelFuture cf = b.connect("127.0.0.1", 8765).sync();

//将5个对象冲刷到服务器端

for(int i = 0; i < 5; i++ ){

Req req = new Req();

req.setId("" + i);

req.setName("pro" + i);

req.setRequestMessage("数据信息" + i);

//读指定的文件

String path = System.getProperty("user.dir") + File.separatorChar + "sources" + File.separatorChar + "001.jpg";

File file = new File(path);

FileInputStream in = new FileInputStream(file);

byte[] data = new byte[in.available()];

in.read(data);

in.close();

req.setAttachment(GzipUtils.gzip(data)); //压缩发送的文件

cf.channel().writeAndFlush(req);

}

cf.channel().closeFuture().sync();

group.shutdownGracefully();

}

}

1. ClientHandler类：与序列化相同

import io.netty.channel.ChannelHandlerAdapter;

import io.netty.channel.ChannelHandlerContext;

import io.netty.util.ReferenceCountUtil;

public class ClientHandler extends ChannelHandlerAdapter{

@Override

public void channelActive(ChannelHandlerContext ctx) throws Exception {

}

@Override

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

Resp resp = (Resp)msg;

System.out.println("Client : " + resp.getId() + ", " + resp.getName() + ", " + resp.getResponseMessage());

}

@Override

public void channelReadComplete(ChannelHandlerContext ctx) throws Exception {

}

@Override

public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws Exception {

ctx.close();

}

}

(7)MarshallingCodeCFactory ：Marshalling工厂与序列化相同

import io.netty.handler.codec.marshalling.DefaultMarshallerProvider;

import io.netty.handler.codec.marshalling.DefaultUnmarshallerProvider;

import io.netty.handler.codec.marshalling.MarshallerProvider;

import io.netty.handler.codec.marshalling.MarshallingDecoder;

import io.netty.handler.codec.marshalling.MarshallingEncoder;

import io.netty.handler.codec.marshalling.UnmarshallerProvider;

import org.jboss.marshalling.MarshallerFactory;

import org.jboss.marshalling.Marshalling;

import org.jboss.marshalling.MarshallingConfiguration;

public final class MarshallingCodeCFactory {

/\*\*

\* 创建Jboss Marshalling解码器MarshallingDecoder

\* @return MarshallingDecoder

\*/

public static MarshallingDecoder buildMarshallingDecoder() {

//首先通过Marshalling工具类的精通方法获取Marshalling实例对象 参数serial标识创建的是java序列化工厂对象。

final MarshallerFactory marshallerFactory = Marshalling.getProvidedMarshallerFactory("serial");

//创建了MarshallingConfiguration对象，配置了版本号为5

final MarshallingConfiguration configuration = new MarshallingConfiguration();

configuration.setVersion(5);

//根据marshallerFactory和configuration创建provider

UnmarshallerProvider provider = new DefaultUnmarshallerProvider(marshallerFactory, configuration);

//构建Netty的MarshallingDecoder对象，俩个参数分别为provider和单个消息序列化后的最大长度

MarshallingDecoder decoder = new MarshallingDecoder(provider, 1024 \* 1024 \* 1);

return decoder;

}

/\*\*

\* 创建Jboss Marshalling编码器MarshallingEncoder

\* @return MarshallingEncoder

\*/

public static MarshallingEncoder buildMarshallingEncoder() {

final MarshallerFactory marshallerFactory = Marshalling.getProvidedMarshallerFactory("serial");

final MarshallingConfiguration configuration = new MarshallingConfiguration();

configuration.setVersion(5);

MarshallerProvider provider = new DefaultMarshallerProvider(marshallerFactory, configuration);

//构建Netty的MarshallingEncoder对象，MarshallingEncoder用于实现序列化接口的POJO对象序列化为二进制数组

MarshallingEncoder encoder = new MarshallingEncoder(provider);

return encoder;

}

}