**使用Netty搭建TCP服务器**

# 一、TCP粘包和拆包

**1.1 粘包和拆包的概念**

TCP编程中，无论是服务端还是客户端，当我们读取或者是发送消息的时候，都需要考虑TCP底层的粘包和拆包机制。

众所周知，TCP是个“流”协议，所谓流，就是没有界限的一串数据。比如河里的水，它们是连成一片的，其间并没有分界线。再加上网络上MTU的往往小于在应用处理的消息数据，所以就会引发一次接收的数据无法满足消息的需要，导致粘包的存在。

TCP底层并不了解上层业务数据的具体含义，它会根据TCP缓冲区的实现情况进行包的拆分，所以在业务上认为，一个完整的包可能会被TCP拆成多个包进行发送，也有可能把多个小的包封装成一个大的数据包发送，这就是所谓的TCP粘包和拆包问题。

处理粘包的唯一方法就是制定应用层的数据通讯协议，通过协议来规范现有接收的数据是否满足消息数据的需要。

## 1.2、Netty中粘包和拆包解决方案

1、消息定长，报文大小固定长度，不够空格补全，发送和接收方遵循相同的约定，这样即使粘包了通过接收方编程实现获取定长报文也能将不同的保温区分开来。

2、包尾添加特殊分隔符，例如每条报文结束都添加回车换行符或者其他的指定特殊字符作为报文分隔符，接收方通过特殊分隔符切分报文区分。

3、将消息分为消息头和消息体，消息头中包含表示消息总长度（或者消息体长度）的字段，通常设计思路为消息头的第一个字段使用int来表示消息的总长度

Netty中对于上面的解决方案已有一些解码器的封装，比如定长解码器FixedLengthFrameDecoder、分隔符解码器DelimiterBasedFrameDecoder、基于回车换行符的解码器LineBasedFrameDecoder等等，在开发时可以直接调用这些编解码器，可以大大提高开发的速度。

# 二、Netty基本介绍

## 2.1 简介

Netty是基于Java NIO client-server的网络应用框架，使用Netty可以快速开发网络应用，例如服务器和客户端协议。Netty提供了一种新的方式来开发网络应用程序，这种新的方式使它很容易使用和具有很强的扩展性。Netty的内部实现是很复杂的，但是Netty提供了简单易用的API从网络处理代码中解耦业务逻辑。Netty是完全基于NIO实现的，所以整个Netty都是异步的。

## 2.2、Netty ByteBuf

当我们进行数据传输的时候，往往需要使用到缓冲区，常用的缓冲区就是JDK NIO类库提供的java.nio.Buffer。从功能而言，ByteBuffer完全可以满足NIO编程的需要，但是NIO编程过于复杂，也存在局限性。ByteBuffer长度固定，一旦分配完成，不可动态修改。JDK

由于只有一个位置指针用于处理读写操作，因此每次读写的时候都需要额外调用flip()，否则功能将出错。

Netty ByteBuf通过内置的复合缓冲类型实现了零拷贝、支持自定义缓冲区的类型、实现了java几种基本类型数据的响应缓冲类型、并且提供了两个指针用于支持顺序读取和写入操作：readerIndex用于标识读取索引，writerIndex用于标识写入索引。两个位置指针将ByteBuf缓冲区分割成三个区域。

（1） readerIndex到writerIndex之间的空间为可读的字节缓冲区

（2） writerIndex到capacity之间为可写的字节缓冲区

（3） 0到readerIndex之间是已经读取过的缓冲区

下面是Netty ByteBuf的常用的几个方法：

isReadable: 缓冲区是否可读

markReaderIndex: 记录当前缓冲区读指针的位置

resetReaderIndex: 重置缓冲区至markReaderIndex的位置处（markReaderIndex的初始位置为0）

readerIndex: 当前缓冲区读指针的位置

readableBytes: 当前缓冲区可读的字节数

read\*: 读取数据

写方法和上面类似

备注: 当一开始使用resetReaderIndex重置缓冲区读指针的位置时候，读指针会被重置0；

当使用markReaderIndex记录过当前缓冲区读指针的位置时，再使用resetReaderIndex重置缓冲区读指针的位置，读指针会被重置到markReaderIndex记录的位置处。

# 三、TCP服务开发

## 3.1自定义协议封装类

/\*\*

\* +------------+------------+------------+------------+

\* |协议开始标志| 长度 | 数据 | 校验位(异或)|

\* +------------+------------+------------+------------+

\* | 0x7F | 2字节 | 数据 | 1字节 |

\* +------------+------------+------------+------------+

\*/

public class SmartProtocol {

private static final byte HEAD\_DATA = 0x7F;

private short contentLength;

private byte[] content;

private byte checkBit;

public SmartProtocol(short contentLength, byte[] content, byte checkBit) {

this.contentLength = contentLength;

this.content = content;

this.checkBit = checkBit;

}

}

## 3.2、自定义协议的编码器

public class SmartEncoder extends MessageToByteEncoder<SmartProtocol> {

@Override

protected void encode(ChannelHandlerContext channelHandlerContext, SmartProtocol smartProtocol, ByteBuf byteBuf) throws Exception {

//1、写入消息头开始标志

byteBuf.writeByte(SmartProtocol.getHeadData());

//2、写入消息长度

byteBuf.writeShort(smartProtocol.getContentLength());

//3、写入数据

byteBuf.writeBytes(smartProtocol.getContent());

//4、写入校验位数据

byteBuf.writeByte(smartProtocol.getCheckBit());

}

}

## 3.3、自定义协议的解码器

public class SmartDecoder extends ByteToMessageDecoder {

public final int BASE\_LENGTH = 1 + 2;

@Override

protected void decode(ChannelHandlerContext channelHandlerContext, ByteBuf byteBuf, List<Object> list) throws Exception {

此处逻辑是按字节一个一个的解析，并匹配，最后检查校验位，通过则将解析的数据封装成实体，以待handler处理

}

}

}

## 3.4、服务端加入协议的编/解码器

@Service("neoChannelInitializer")

public class NeoChannelInitializer extends ChannelInitializer<SocketChannel> {

……

@Override

protected void initChannel(SocketChannel ch) throws Exception {

1. 每次请求获取一个新的handler
2. 将编解码器放入pipeline
3. 如果有需要，可以将心跳的handler放入pipeline
4. 最后将自定义的业务handler放入pipeline

}

}

## 3.5、自定义Handler

public class TcpServerHandler extends ChannelHandlerAdapter {

@Override

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

异常处理

}

@Override

public void userEventTriggered(ChannelHandlerContext ctx, Object evt) throws Exception {

读写超时、心跳处理

}

}

@Override

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

具体的业务逻辑分发部分处理

}

}

# 四、代码优化

ChannelOption.SO\_BACKLOG --等待队列的大小

ChannelOption.SO\_KEEPALIVE, false --是否保持长连接

ChannelOption.CONNECT\_TIMEOUT\_MILLIS --客户端连接超时时间

ChannelOption.WRITE\_BUFFER\_HIGH\_WATER\_MARK, 64 \* 1024 //水位高值

ChannelOption.WRITE\_BUFFER\_LOW\_WATER\_MARK, 32 \* 1024 //水位低值，设置后可判断buffer的isWritable是false还是true，从而控制读写，防止buffer不断地增长占用太多系统资源。所以应用应该判断isWritable

ChannelOption.RCVBUF\_ALLOCATOR, AdaptiveRecvByteBufAllocator.DEFAULT //容量可自动动态调整的接收缓冲区分配器，减少内存使用

pipeline.addLast("ping", new IdleStateHandler(5, 0, 0)); 心跳监测0表示关闭 读超时:5s内没有数据接收，写超时:没有数据发送 全部空闲时间:没有数据接收或者发送

# 五、netty和spring整合问题

要点：对于每一个客户端，我们都是new一个handler处理，所以在整合Spring的时候一定要注意，每次新的处理请求，一定要重新创建一个handler实例：tcpServerHandler必须是每次请求都重新创建一个，底层pipiline不是可共享的,否则多次请求下将报错。

在Spring中有一个设置bean的scope为prototype表示多实例模式。这里一定要主要Spring的prototype模式的坑， prototype作用域的对象，每次getBean

我的实现：

<bean id="tcpServerHandler" class="com.neo.agps.tcp.server.TcpServerHandler" scope="prototype"/>

获取handler时，实现spring的ApplicationContextAware接口的setApplicationContext方法；

每次需要handler都这样：context.getBean("tcpServerHandler"));

# 六、server的自启动

这里还是使用Spring的扩展，tcpserver实现ApplicationListener<ContextRefreshedEvent>接口

然后实现onApplicationEvent方法：

@Override

public void onApplicationEvent(ContextRefreshedEvent evt) {

if (evt.getApplicationContext().getParent() == null) {

...

}

}

注意一定要判断父容器，否则可能会启动两次