## netty与常见通讯协议应用

Netty 是一个利用 Java 的高级网络的能力，隐藏其背后的复杂性而提供一个易于使用的 API 的客户端/服务器框架

1 . Netty可以来做什么

阿里分布式服务框架 Dubbo 的 RPC 框架使用 Dubbo 协议进行节点间通信，Dubbo 协议默认使用 Netty 作为基础通信组件，用于实现各进程节点之间的内部通信。

淘宝的消息中间件 RocketMQ 的消息生产者和消息消费者之间，也采用 Netty 进行高性能、异步通信。

游戏行业：无论是手游服务端、还是大型的网络游戏，Java 语言得到了越来越广泛的应用。Netty 作为高性能的基础通信组件，它本身提供了 TCP/UDP 和 HTTP 协议栈，非常方便定制和开发私有协议栈。账号登陆服务器、地图服务器之间可以方便的通过 Netty 进行高性能的通信。

大数据领域：经典的 Hadoop 的高性能通信和序列化组件 Avro 的 RPC 框架，默认采用 Netty 进行跨节点通信，它的 Netty Service 基于 Netty 框架二次封装实现。

通信行业：Netty 的异步高性能、高可靠性和高成熟度的优点，使它在通信行业得到了大量的应用。

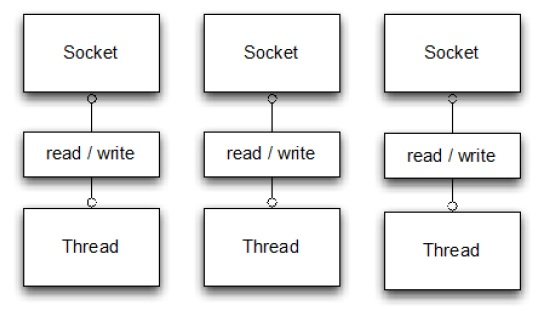
2. Netty的优点

并发高，传输快，封装好

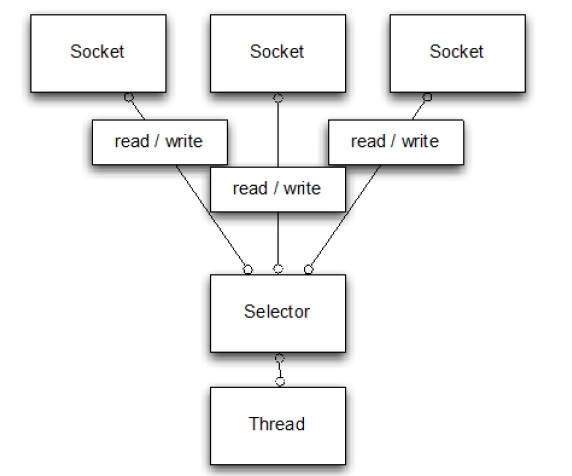
3 . nett为什么并发高--Netty的模式

Netty是一款基于NIO（Nonblocking I/O，非阻塞IO）开发的网络通信框架，对比于BIO（Blocking I/O，阻塞IO），他的并发性能得到了很大提高。

传统的BIO通信方式



NIO的通信方式



可以通过上面两图看出，NIO相对于BIO可以用更少的线程，处理更多的的链接。

在BIO中，等待客户端发数据这个过程是阻塞的，这样就造成了一个线程只能处理一个请求的情况，而机器能支持的最大线程数是有限的，这就是为什么BIO不能支持高并发的原因。

在NIO中，当一个Socket建立好之后，Thread并不会阻塞去接受这个Socket，而是将这个请求交给Selector，Selector会不断的去遍历所有的Socket，一旦有一个Socket建立完成，他会通知Thread，然后Thread处理完数据再返回给客户端——*这个过程是阻塞的*，这样就能让一个Thread处理更多的请求了。

4. netty为什么传输快—netty零拷贝

Netty中也用到了FileChannel.transferTo方法，所以Netty的零拷贝也包括上面将的操作系统级别的零拷贝。除此之外，在ByteBuf的实现上，Netty也提供了零拷贝的一些实现。

关于ByteBuffer，Netty提供了两个接口:  
1. ByteBuf  
2. ByteBufHolder

对于ByteBuf，Netty提供了多种实现：  
1. Heap ByteBuf:直接在堆内存分配  
2. Direct ByteBuf：直接在内存区域分配而不是堆内存  
3. CompositeByteBuf：组合Buffer

5.netty为什么封装好？

**一 基于netty开发tcp自定义协议服务器**

**拆包和粘包**

UCP是基于报文发送的，UDP报文的首部会有16bit来表现UDP数据的长度，所以不同的报文之间是可以区别隔离出来的，所以应用层接收传输层的报文时，不会存在拆包和粘包的问题；

而TCP是基于字节流传的，应用层和传输层之前数据交互是大小不等的数据块，但TCP对这些数据块只是一连串的数据流，它并不知道哪些数据块跟哪些数据块是该一起发，哪个数据块是应该单独一块的，因为TCP并没有像UDP那样首部有数据长度，所以TCP存在拆包和粘包的问题。

**原因**

1.要发送的数据包大于发送缓存区的可用空间时，数据被会拆包；

2.要发送的数据大于MSS（Maximum Segment Size最大报文段），TCP在发送前会对数据进行拆包；

TCP在三次握手建立连接过程中，会在SYN报文中使用MSS选项功能，协商交互双方能够接收的最大段长MSS值。

MSS是传输层TCP协议范畴内的概念，它是标识TCP能够承载的最大的应用数据段长度，因此，MSS=MTU-20字节TCP报头-20字节IP报头，那么在以太网环境下，MSS值一般就是1500-20-20=1460字节。

3.发送的数据小于缓存区，则TCP会将几次的数据一次性发送，会存在粘包；

**拆包、粘包举例**

比如发送端要往接收端发送2个数据包

1.收到2个数据包，没发送拆包和粘包情况；

2.收到1个数据包，TCP把2个数据包合成1个发送给接收端了，这样应用层不能处理合成1个的两个数据包，应用层不知道两个数据包之间的分隔在哪，所以很难处理，这是粘包问题；

3.收到2个数据包，但1个数据包产生了粘包（发送端的1个半数据包），另1个数据包产生了拆包（只有发送端中1个数据包的半个包），这样应用层也是很难处理粘包、拆包的；

**解决**

由于TCP是不知道应用层的数据包的分界的，所以我们应用层是决定不了传输层对数据包拆包还是粘包，不过应用层可以通应用层协议栈设计给数据包加分界标记，来处理最后接收到的数据，不管拆分还是粘包都可以处理好。

1.发送端给数据包增加首部，首部包含数据包中数据的长度，这样接收端的应用层接收数据后，根据首部中的长度就知道数据的实际长度了，可以很好处理数据了。

通常设计思路，比如第1个字段使用32int表示数据的长度，接着是数据内容。

2.设置数据包的长度为固定的长度，不够数据则以空格填补；

3.应用层在发送每个数据包时，给每个数据包加分界标记，比如回车换行

**Netty 中解决拆包与粘包**

**netty 提供的解码器**

　　DelimiterBasedFrameDecoder 解决TCP的粘包**解码**器

　　StringDecoder 消息转成String**解码**器

　　LineBasedFrameDecoder 自动完成标识符分隔**解码**器

　　FixedLengthFrameDecoder 固定长度**解码**器，二进制

　　Base64Decoder base64 **解码**器

**netty 提供的编码器**

　　 Base64Encoder base64**编码**器

　　StringEncoder 消息转成String**编码**器

　　LineBasedFrameDecoder 自动完成标识符分隔**编码**器  
　　MessageToMessageEncoder 根据 消息对象 编码为消息对象

自定义协议格式如下

|  |  |  |
| --- | --- | --- |
| 参数 | 字节长度 | 说明 |
| Type | 1 | 消息类型 |
| qos | 1 | 消息等级 |
| length | 4 | 消息长度的长度 |
| content | 无固定长度 | 消息 |

该协议通过LengthFieldBasedFrameDecoder来解码，该类的实现方式是上面的解决方案1

LengthFieldBasedFrameDecoder有类有4个参数分别如下：

      
 param maxFrameLength  帧的最大长度  
 param lengthFieldOffset length字段偏移的地址

param lengthFieldLength length字段所占的字节长  
param lengthAdjustment 修改帧数据长度字段中定义的值，可以为负数 因为有时 候我们习惯把头部记入长度,若为负数,则说明要推后多少个字段  
param initialBytesToStrip 解析时候跳过多少个长度  
param failFast 为true，当frame长度超过maxFrameLength时立即报TooLongFrameException异常，为false，读取完整个帧再报异

通过上面4个参数我们可以完美解析我们上面的自定义协议，param maxFrameLength 我们设置我们消息的最大长度根据自己情况来定，

param lengthFieldOffset length 这个偏移字段是指我们设置消息长度之前前面的字段空间，我们自定义协议type和flag各占一个字节，所以我们需要偏移的量是2。

param lengthFieldLength length 消息长度是4，我们设置该消息长度是4，读完该字节后就是消息体

param lengthAdjustment是指我们设置固定包头的情况下，我们需要设置的偏移量，因为前面我们没有设置包头，固设置为0

param initialBytesToStrip 如果设置了包头我们就需要在此填写包头的长度，我们没设置固设置为0；

**具体实现：**

**见附件**

**二 基于netty开发http协议服务器**

**1 http协议**

**请求包**

****

主要包含三部分：请求行(line)，请求头(header)，请求正文(body)

请求行(Line)：主要包含三部分：Method ，URI ，协议/版本。 各部分之间使用空格(SP)分割。整个请求头使用CRLF分割。(比如：POST /1.0.0/\_health\_check HTTP/1.1 CRLF)

请求头(Header)： 格式为(name ：value)，用于客户端请求的描述信息。header之间以CRLF进行分割。最后一个header会多加一个CRLF。( 比如：Connection: keep-alive CRLF CRLF)

请求正文(body) ：里面主要是Post提交的数据(可支持多种格式，格式在Content-Type定义，长度是在Content-Length里面定义)。

**相应包**

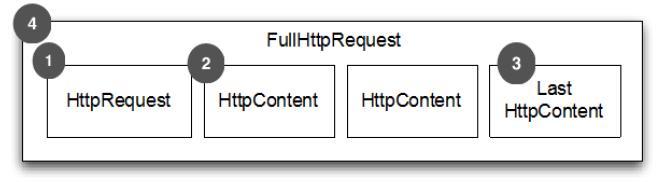
主要包含三部分：状态行(line),响应头(header),响应正文(body)

状态行(line)：包含三部分：http版本，服务器返回状态码，描述信息。以CRLF进行分割。 ( 比如：HTTP/1.1 200 OK CRLF)

响应头(header) ： 格式为(name ：value)，用于服务器返回的描述信息。header之间以CRLF进行分割。最后一个header会多加一个CRLF (比如：Content-Type: text/html CRLF Content-Encoding:gzip CRLF CRLF)

响应正文(body)：里面主要是返回数据(可支持多种格式，格式在Content-Type定义，长度是在Content-Length里面定义)。

Netty中http请求组成



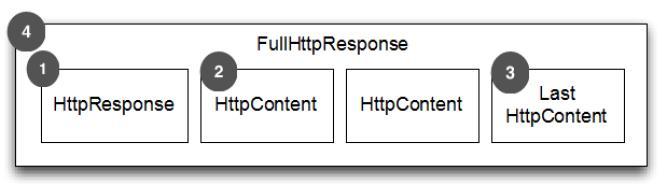
 HTTP Request 第一部分是包含的头信息

 HttpContent 里面包含的是数据，可以后续有多个 HttpContent 部分

 LastHttpContent 标记是 HTTP request 的结束，同时可能包含头的尾部信息

 完整的 HTTP request，由1，2，3组成

Netty中http响应组成



 HTTP response 第一部分是包含的头信息

 HttpContent 里面包含的是数据，可以后续有多个 HttpContent 部分

 LastHttpContent 标记是 HTTP response 的结束，同时可能包含头的尾部信息

 完整的 HTTP response，由1，2，3组成

2 netty中编写一个简单的http服务器（基于tcp短连接）

服务启动配置类

public class HttpServer {  
 private final int port =8080;  
 public static void main(String[] args) throws Exception {  
 new HttpServer().start();  
 }  
 public void start() throws Exception {  
 ServerBootstrap b = new ServerBootstrap();  
 NioEventLoopGroup boss = new NioEventLoopGroup();//接收请求  
 NioEventLoopGroup work = new NioEventLoopGroup();//处理接收的请求  
 b.group(boss,work)  
 .channel(NioServerSocketChannel.class)  
 .childHandler(new ChannelInitializer<SocketChannel>() {  
 @Override  
 public void initChannel(SocketChannel ch)  
 throws Exception {  
 System.*out*.println("initChannel ch:" + ch);  
 ch.pipeline()  
 .addLast("decoder", new HttpRequestDecoder()) // 1 http请求解码  
 .addLast("encoder", new HttpResponseEncoder()) // 2 http响应编码  
 .addLast("aggregator", new HttpObjectAggregator(512 \* 1024)) // 3 聚合器  
 .addLast("handler", new HttpProcessHandle()); // 4 处理类  
 }  
 })  
 .option(ChannelOption.*SO\_BACKLOG*, 128) // determining the number of connections queued  
 .childOption(ChannelOption.*SO\_KEEPALIVE*, Boolean.*TRUE*);  
 b.bind(port).sync();  
 }  
}

消息处理类

public class HttpProcessHandle extends SimpleChannelInboundHandler<FullHttpRequest> {  
  
 private AsciiString contentType = HttpHeaderValues.*TEXT\_PLAIN*;  
 private String result="";  
  
 @Override  
 protected void channelRead0(ChannelHandlerContext ctx, FullHttpRequest msg) throws Exception {  
 System.*out*.println("class:" + msg.getClass().getName());  
  
 if(! (msg instanceof FullHttpRequest)){  
 result="未知请求!";  
 send(ctx,result,HttpResponseStatus.*BAD\_REQUEST*);  
 return;  
 }  
 FullHttpRequest httpRequest = (FullHttpRequest)msg;  
 try{  
 String path=httpRequest.uri(); //获取路径  
 String body = getBody(httpRequest); //获取参数  
 HttpMethod method=httpRequest.method();//获取请求方法  
 //如果不是这个路径，就直接返回错误  
 if(!"/test".equalsIgnoreCase(path)){  
 result="非法请求!";  
 send(ctx,result,HttpResponseStatus.*BAD\_REQUEST*);  
 return;  
 }  
 System.*out*.println("接收到:"+method+" 请求");  
 //如果是GET请求  
 if(HttpMethod.*GET*.equals(method)){  
 //接受到的消息，做业务逻辑处理...  
 System.*out*.println("body:"+body);  
 result="GET请求";  
 send(ctx,result,HttpResponseStatus.*OK*);  
 return;  
 }  
 //如果是POST请求  
 if(HttpMethod.*POST*.equals(method)){  
 //接受到的消息，做业务逻辑处理...  
 System.*out*.println("body:"+body);  
 result="POST请求";  
 send(ctx,result,HttpResponseStatus.*OK*);  
 return;  
 }  
  
 //如果是PUT请求  
 if(HttpMethod.*PUT*.equals(method)){  
 //接受到的消息，做业务逻辑处理...  
 System.*out*.println("body:"+body);  
 result="PUT请求";  
 send(ctx,result,HttpResponseStatus.*OK*);  
 return;  
 }  
 //如果是DELETE请求  
 if(HttpMethod.*DELETE*.equals(method)){  
 //接受到的消息，做业务逻辑处理...  
 System.*out*.println("body:"+body);  
 result="DELETE请求";  
 send(ctx,result,HttpResponseStatus.*OK*);  
 return;  
 }  
 }catch(Exception e){  
 System.*out*.println("处理请求失败!");  
 e.printStackTrace();  
 }finally{  
 //释放请求  
 httpRequest.release();  
 }  
  
  
 DefaultFullHttpResponse response = new DefaultFullHttpResponse(HttpVersion.*HTTP\_1\_1*,  
 HttpResponseStatus.*OK*,  
 Unpooled.*wrappedBuffer*("test".getBytes())); // 2  
  
 HttpHeaders heads = response.headers();  
 heads.add(HttpHeaderNames.*CONTENT\_TYPE*, contentType + "; charset=UTF-8");  
 heads.add(HttpHeaderNames.*CONTENT\_LENGTH*, response.content().readableBytes()); // 3  
 heads.add(HttpHeaderNames.*CONNECTION*, HttpHeaderValues.*KEEP\_ALIVE*);  
  
 ctx.write(response);  
 }  
  
  
 */\*\*  
 \* 发送的返回值  
 \** ***@param*** *ctx 返回  
 \** ***@param*** *context 消息  
 \** ***@param*** *status 状态  
 \*/* private void send(ChannelHandlerContext ctx, String context,HttpResponseStatus status) {  
 FullHttpResponse response = new DefaultFullHttpResponse(HttpVersion.*HTTP\_1\_1*, status, Unpooled.*copiedBuffer*(context, CharsetUtil.*UTF\_8*));  
 response.headers().set(HttpHeaderNames.*CONTENT\_TYPE*, "text/plain; charset=UTF-8");  
 ctx.writeAndFlush(response).addListener(ChannelFutureListener.*CLOSE*);  
 }  
  
 */\*\*  
 \* 获取body参数  
 \** ***@param*** *request  
 \** ***@return*** *\*/* private String getBody(FullHttpRequest request){  
 ByteBuf buf = request.content();  
 return buf.toString(CharsetUtil.*UTF\_8*);  
 }  
 @Override  
 public void channelReadComplete(ChannelHandlerContext ctx) throws Exception {  
 System.*out*.println("channelReadComplete");  
 super.channelReadComplete(ctx);  
 ctx.flush(); // 4  
 }  
  
 @Override  
 public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws Exception {  
 System.*out*.println("exceptionCaught");  
 if(null != cause) cause.printStackTrace();  
 if(null != ctx) ctx.close();  
 }  
}

三 开发基于netty与mqtt协议的消息服务器（基于tcp长连接）

1 mqtt协议

MQTT是基于TCP长连接，首先是先跟MQTT服务器建立TCP连接，然后发送登录请求，要保持长连接，还要定时发心跳包跟服务保持连接。

Mqtt的基本消息类型



2 netty中基于mqtt协议编写一个简单的即时通讯服务器

服务启动类

public class MqttServer {  
 public static void main(String[] args) throws Exception {  
 new MqttServer().start();  
 }  
 void start(){  
 ServerBootstrap b = new ServerBootstrap();  
 NioEventLoopGroup m\_bossGroup = new NioEventLoopGroup();//接收请求  
 NioEventLoopGroup m\_workerGroup = new NioEventLoopGroup();//处理接收的请求  
 try {  
 b.group(m\_bossGroup, m\_workerGroup).channel(ServerSocketChannel.class)  
 .childHandler(new ChannelInitializer<SocketChannel>() {  
 @Override  
 public void initChannel(SocketChannel ch) throws Exception {  
 ChannelPipeline pipeline = ch.pipeline();  
 pipeline.addLast("decoder", new MqttDecoder(2048)); //解码  
 pipeline.addLast("encoder", MqttEncoder.*INSTANCE*);//编码  
 pipeline.addLast("handler", new NettyMQTTHandler());  
 }  
 })  
 .option(ChannelOption.*SO\_BACKLOG*, 512)//缓存  
 .option(ChannelOption.*SO\_REUSEADDR*, true)  
 .childOption(ChannelOption.*TCP\_NODELAY*, true)  
 .childOption(ChannelOption.*SO\_KEEPALIVE*, true).bind(6060).sync();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
}

消息的处理

public void channelRead(ChannelHandlerContext ctx, Object message) throws Exception {  
 MqttMessage msg = (MqttMessage) message;  
 if (msg.fixedHeader() == null) {  
 throw new IOException("Unknown packet");  
 }  
 MqttMessageType messageType = msg.fixedHeader().messageType();  
  
 try {  
 switch (messageType) {  
 case *CONNECT*:  
// m\_processor.processConnect(ctx.channel(), (MqttConnectMessage) msg);  
 break;  
 case *SUBSCRIBE*:  
// m\_processor.processSubscribe(ctx.channel(), (MqttSubscribeMessage) msg);  
 break;  
 case *UNSUBSCRIBE*:  
// m\_processor.processUnsubscribe(ctx.channel(), (MqttUnsubscribeMessage) msg);  
 break;  
 case *PUBLISH*:  
// m\_processor.processPublish(ctx.channel(), (MqttPublishMessage) msg);  
 break;  
 case *PUBREC*:  
// m\_processor.processPubRec(ctx.channel(), msg);  
 break;  
 case *PUBCOMP*:  
// m\_processor.processPubComp(ctx.channel(), msg);  
 break;  
 case *PUBREL*:  
// m\_processor.processPubRel(ctx.channel(), msg);  
 break;  
 case *DISCONNECT*:  
// m\_processor.processDisconnect(ctx.channel());  
 break;  
 case *PUBACK*:  
// m\_processor.processPubAck(ctx.channel(), (MqttPubAckMessage) msg);  
 break;  
 case *PINGREQ*:  
 MqttFixedHeader pingHeader = new MqttFixedHeader(  
 MqttMessageType.*PINGRESP*,  
 false,  
 *AT\_MOST\_ONCE*,  
 false,  
 0);  
 MqttMessage pingResp = new MqttMessage(pingHeader);  
 ctx.writeAndFlush(pingResp).addListener(*CLOSE\_ON\_FAILURE*);  
 break;  
 default:  
  
 break;  
 }  
 } catch (Throwable ex) {  
  
 ctx.channel().close().addListener(new ChannelFutureListener() {  
  
 public void operationComplete(ChannelFuture channelFuture) throws Exception {  
  
 }  
 });  
 } finally {  
 ReferenceCountUtil.*release*(msg);  
 }

四 基于netty与udp的消息服务器（无连接）

1 udp简介

UDP是User Datagram Protocol的简称，中文名是用户数据报协议，是OSI参考模型中的传输层协议，它是一种无连接的传输层协议，提供面向事务的简单不可靠信息传送服务。

由于缺乏可靠性且属于非连接导向协议，UDP的应用一般必须允许一定量的丢包、出错和复制粘贴。但有些应用，比如TFTP，需要可靠性保证，则必须在应用层增加根本的可靠机制。但是绝大多数UDP应用都不需要可靠机制，甚至可能因为引入可靠机制而降低性能。流媒体、即时多媒体游戏和IP电话（VoIP）就是典型的UDP应用。如果某个应用需要很高的可靠性，那么可以用传输控制协议（即TCP协议）来代替UDP。

使用UDP协议的应用有：域名系统（DNS）、简单网络管理协议（SNMP）、动态主机配置协议（DHCP）、路由信息协议（RIP）等等。因为UDP不属于连接型协议，因而具有资源消耗小，处理速度快的优点，所以通常音频、视频和普通数据在传送时使用UDP较多，因为它们即使偶尔丢失几个数据包，也不会对接收结果产生太大影响。

2 netty中编码一个简单的udp通讯服务器

启动配置类

public class UdpServer {  
 public static void main(String[] args) throws Exception {  
 new UdpServer().start();  
 }  
 void start(){  
 EventLoopGroup eventLoopGroup = new NioEventLoopGroup();  
 try {  
 Bootstrap bootstrap = new Bootstrap();  
 bootstrap.group(eventLoopGroup)  
 .channel(NioDatagramChannel.class)  
 .option(ChannelOption.*SO\_BROADCAST*,true)  
 .handler(new UdpServerHandler());  
 bootstrap.bind(7070).sync().channel().closeFuture().await();  
 } catch (InterruptedException e) {  
  
 }finally {  
 eventLoopGroup.shutdownGracefully();  
 }  
  
 }  
}

消息处理

public class UdpServerHandler extends SimpleChannelInboundHandler<DatagramPacket> {  
 @Override  
 public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws Exception {  
 ctx.close();  
 }  
 protected void channelRead0(ChannelHandlerContext ctx, DatagramPacket datagramPacket) throws Exception {  
 String req = datagramPacket.content().toString(CharsetUtil.*UTF\_8*); //接收的消息  
 ctx.writeAndFlush(new DatagramPacket(Unpooled.*copiedBuffer*(req, CharsetUtil.*UTF\_8*), datagramPacket.sender()));// 将消息发送到对应地址的客户端  
 }  
}