# eNetty权威指南

# Java的I/O演进之路

## 1.1 I/O基础入门

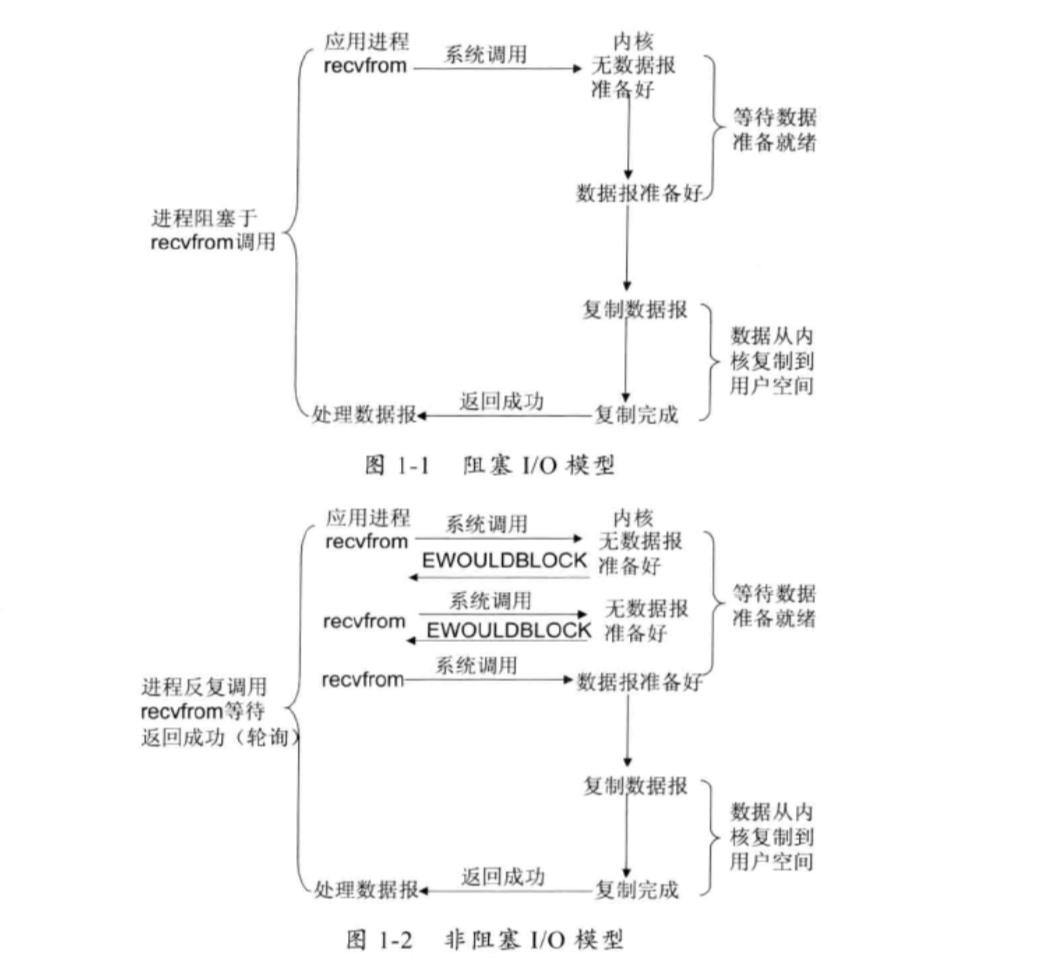
jdk1.4之前，java I/O存在的问题：

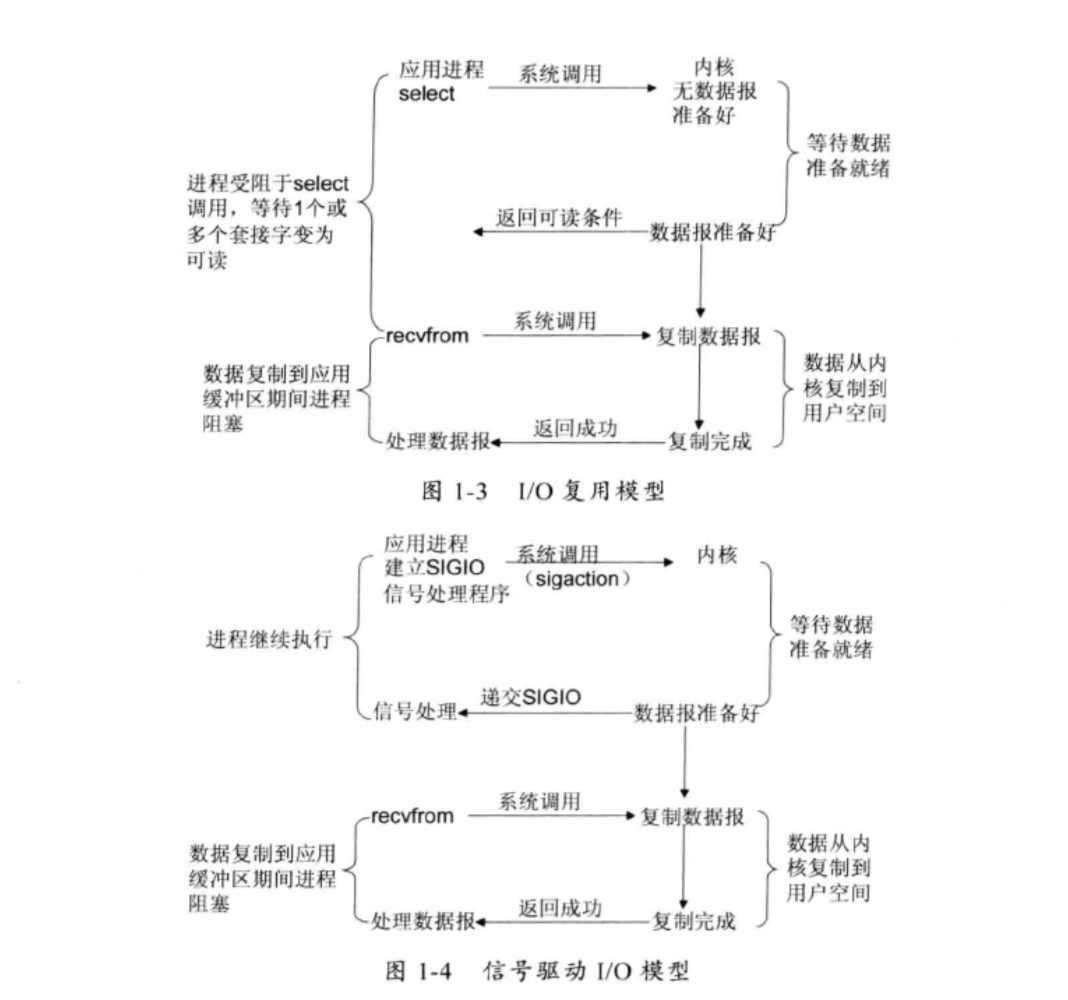
1. 没有数据缓冲区，I/O性能存在问题；
2. 没有C或者C++中的Channel概念，智能输入和输出流
3. 同步阻塞式I/O通信(BIO)，通常会导致通信线程时间阻塞
4. 支持的字符集有限，引荐可移植性不好。

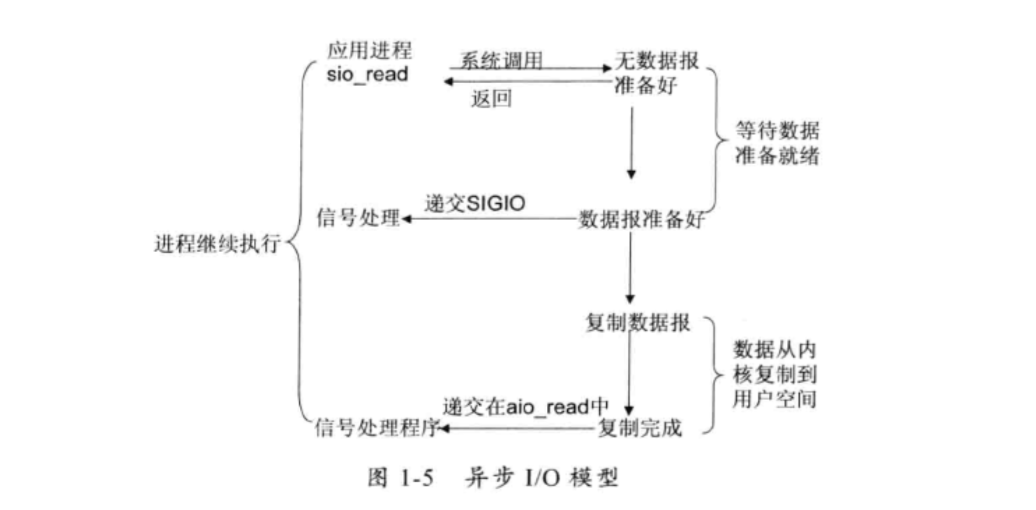
### 1.1.1 Linux网络I/O模型简介

5种IO模型：

1. 阻塞IO模型，
2. 非阻塞IO模型：
3. IO复用模型：
4. 信号驱动模型
5. 异步IO







### 1.1.2 IO多路复用技术

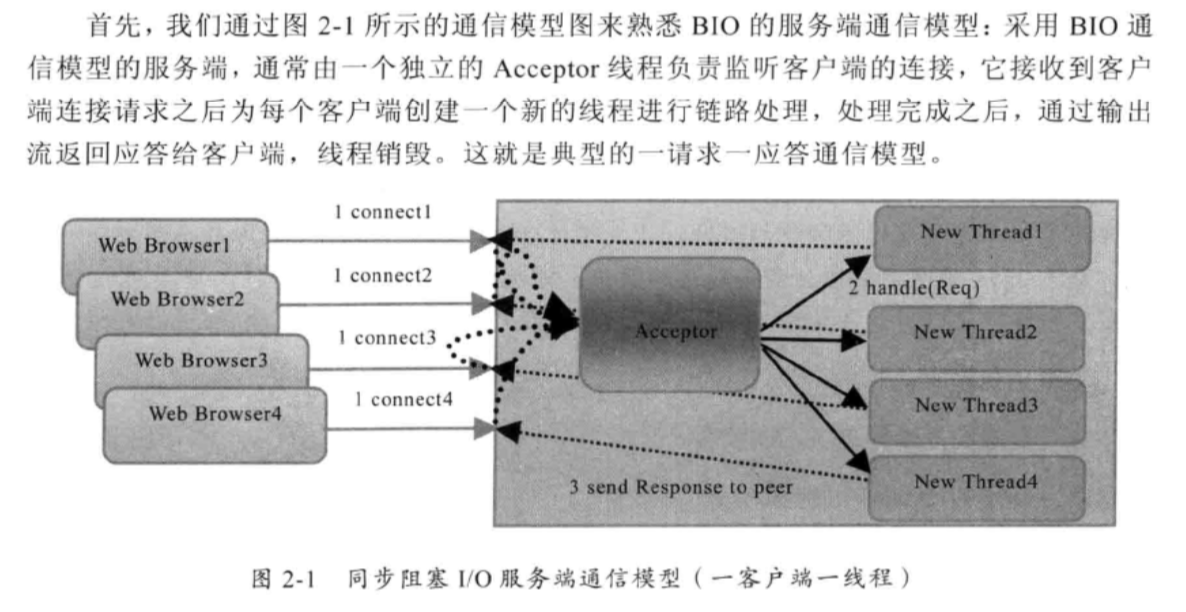
IO多路复用主要应用场景：

1. 服务器同时处理多个处于监听状态或者多个连接状态的套接字
2. 服务器需要同时处理多种网络协议的套接字

# NIO入门

## 2.1 传统的BIO编程

### 2.1.1 BIO通信模型图



TimeServer

|  |
| --- |
| package com.netty.study.chapter02;  import java.io.IOException; import java.net.ServerSocket; import java.net.Socket;  public class TimeServer {  public static void main(String[] args) throws IOException {   int port = 8080;  if (args != null && args.length > 0) {  try {  port = Integer.*valueOf*(args[0]);  } catch (NumberFormatException e) {  }  }   ServerSocket server = null;   try {  server = new ServerSocket(port);  System.*out*.println("The time server is start in port :" + port);  Socket socket = null;  while (true){  socket = server.accept();  new Thread(new TimeServerHandler(socket)).start();  }   } finally {  if(server != null){  System.*out*.println("The time server close");  server.close();  server = null;  }  }   }  } |

TimeServerHandler

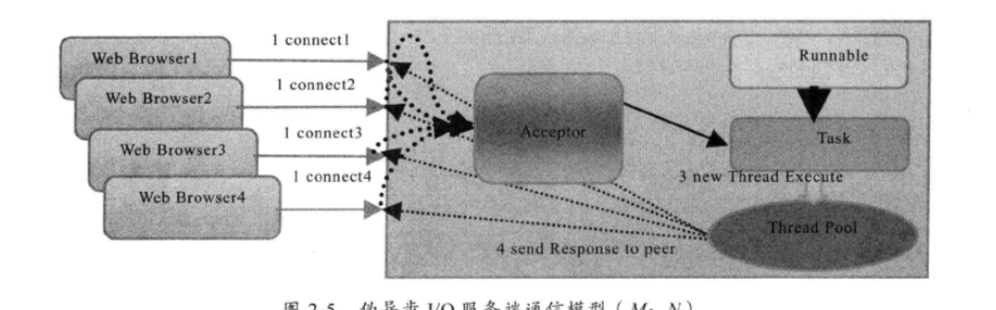
|  |
| --- |
| package com.netty.study.chapter02;  import java.io.BufferedReader; import java.io.IOException; import java.io.InputStreamReader; import java.io.PrintWriter; import java.net.Socket; import java.util.Date;  public class TimeServerHandler implements Runnable {   private Socket socket;   public TimeServerHandler(Socket socket) {  this.socket = socket;  }   public void run() {  BufferedReader in = null;  PrintWriter out = null;   try {  in = new BufferedReader(new InputStreamReader(this.socket.getInputStream()));  out = new PrintWriter(this.socket.getOutputStream(),true);  String currentTime = null;  String body = null;  while (true) {  body = in.readLine();  if (body == null) {  break;  }  System.*out*.println("The time server receive order : " + body);  currentTime = "QUERY TIME ORDER".equalsIgnoreCase(body) ? new Date(System.*currentTimeMillis*()).toString() : "BAD ORDER";  out.println(currentTime);  }  } catch (Exception e) {  if(in != null){  try {  in.close();  }catch (IOException e1){  e1.printStackTrace();  }  }   if(out != null){  out.close();  out = null;  }   if(this.socket != null){  try {  this.socket.close();  }catch (IOException e1){  e.printStackTrace();  }  this.socket = null;  }  }    } } |

TimeCLient

|  |
| --- |
| package com.netty.study.chapter02;  import java.io.BufferedReader; import java.io.IOException; import java.io.InputStreamReader; import java.io.PrintWriter; import java.net.Socket;  public class TimeClient {   public static void main(String[] args) {  int port = 8080;  if (args != null && args.length > 0) {  try {  port = Integer.*valueOf*(args[0]);  } catch (NumberFormatException e) {   }  }   Socket socket = null;  BufferedReader in = null;  PrintWriter out = null;   try {  socket = new Socket("127.0.0.1", port);  in = new BufferedReader(new InputStreamReader(socket.getInputStream()));   out = new PrintWriter(socket.getOutputStream(), true);  out.println("QUERY TIME ORDER");  System.*out*.println("Send order 2 server succeed");  String resp = in.readLine();  System.*out*.println("Now is: " + resp);   } catch (Exception e) {   } finally {  if (out != null){  out.close();  out = null;  }   if(in != null){  try {  in.close();  }catch (IOException e){  e.printStackTrace();  }   in = null;  }    if(socket != null){  try {  socket.close();  }catch (IOException e){  e.printStackTrace();  }  socket = null;  }   }  }  } |

## 2.2 伪异步IO编程

后端通过一个线程池来处理多个客户端的请求接入，形成客户端个数M：线程池最大线程数N的比例关系，其中M可以远远大于N。通过线程池可以灵活调配线程资源。



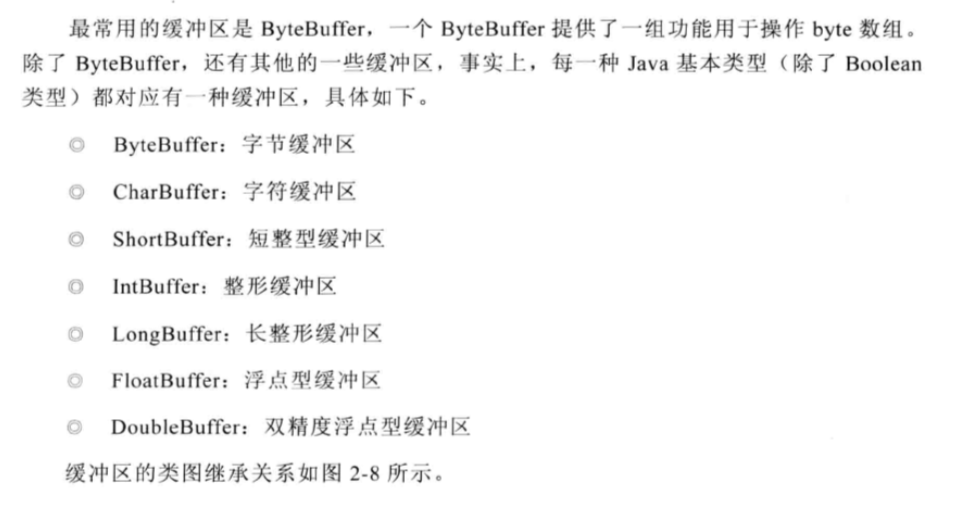
伪异步IO实际上进京对IO线程模型的简单优化，无法从根本上解决同步IO问题。

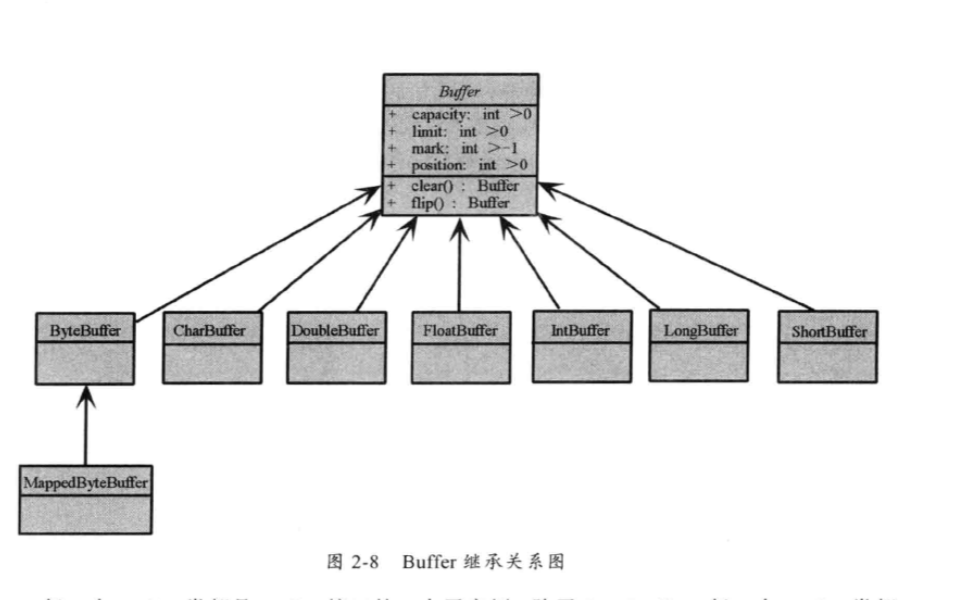
1. 服务端处理缓慢，返回应答消息耗费60s，平时只需要10ms
2. 采用伪异步IO的线程正在读取故障服务节点的响应，由于读取输入流是阻塞的，它将会被同步阻塞60s
3. 由于线程池采用阻塞对列实现，当对垒积满之后，后续入队列的操作将被阻塞。
4. 由于线程池采用阻塞对列实现，当对列积满之后，后续入队列的操作将被阻塞
5. 由于前端只有一个Accptor线程接收客户端接入，它被阻塞在线程池的同步阻塞对列之后，新的客户端请求消息将被拒绝，客户端会发生大量的连接超时
6. 由于几乎所有的连接都超时，调用者会认为系统已经崩溃，无法接收新的请求消息。

## 2.3 NIO编程

### 2.3.1 NIO类库介绍

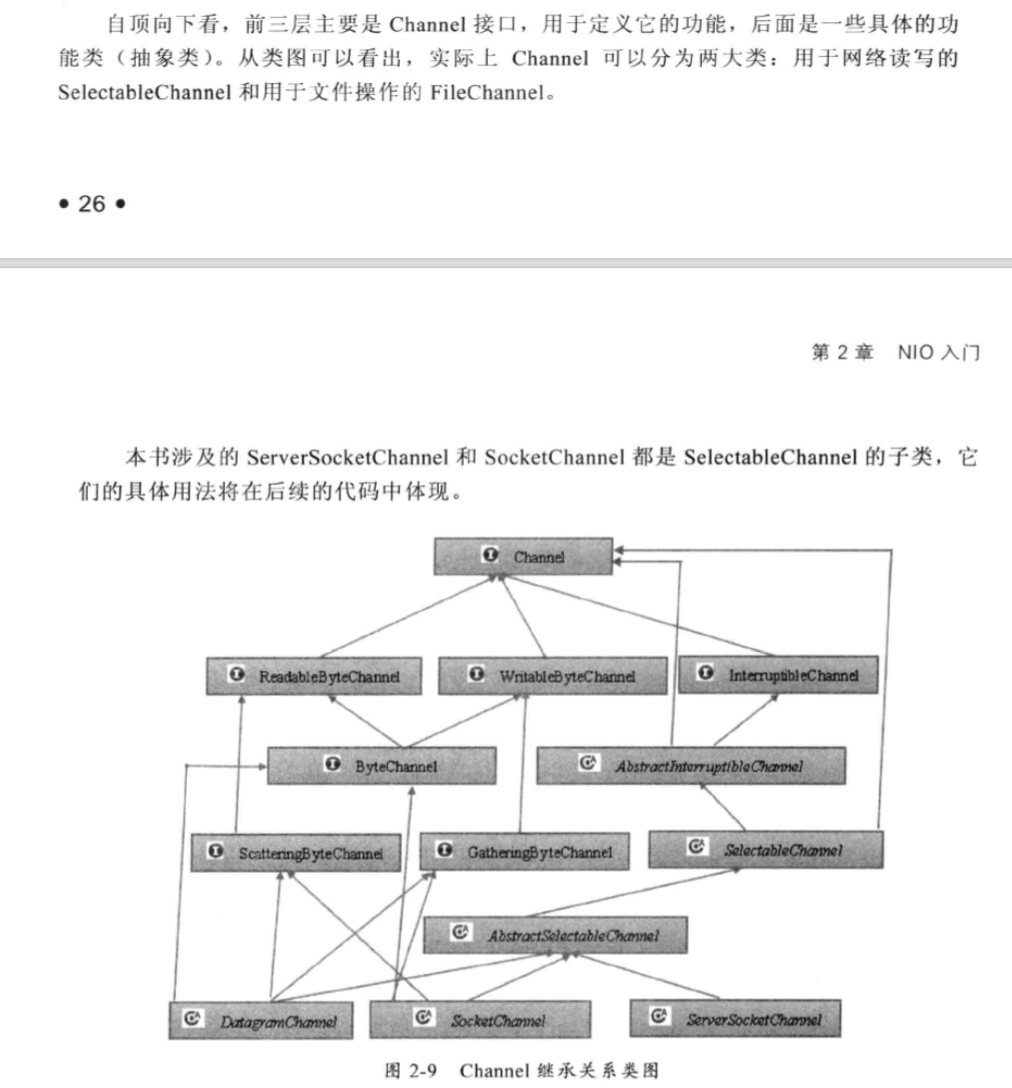
**1 缓冲区Buffer**





**2 通道Channel**

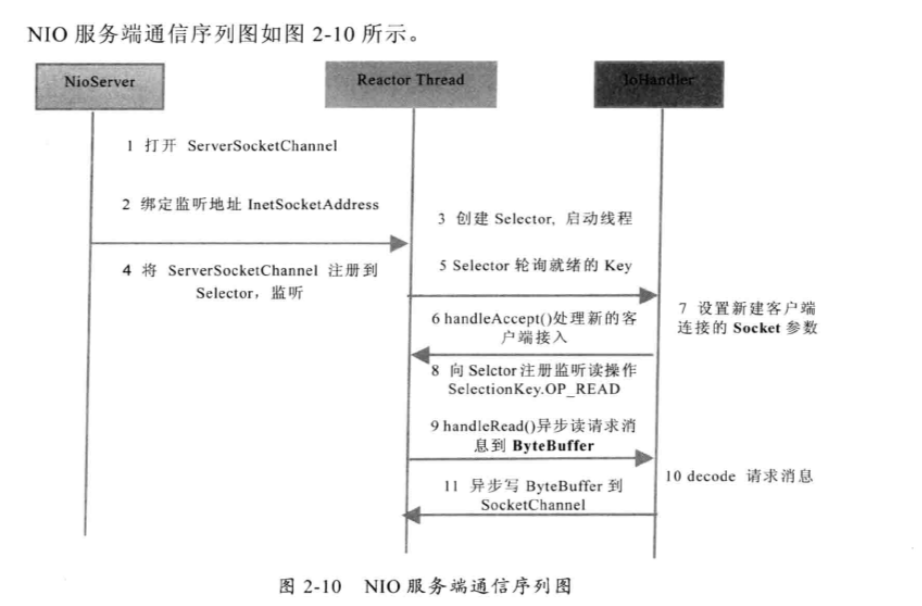
网络数据通过Channel读取和写入。通道与流的不同之处在于通道是双向的，流只是一个方向上移动，通道可以用于读、写或者二者同时进行。



**3 多路复用器Selector**

Selector会不断地轮询注册在其上的Channel，如果Channel上面发生读或者写时间，这个Channel就处于就绪状态，会被Selector轮询出来，然后通过SelectionKey可以获取就绪Channel的集合，进行后续的IO操作。

### 2.3.2 NIO服务端序列图



步骤一：打开ServerSocketChannel，用于监听客户端的连接，它是所有客户端连接的父管道，

ServerSocketChannel acceptorSvr = ServerSocketChannel.*open*();

步骤二：丙丁监听端口，设置连接为非主色模式，

acceptorSvr.socket().bind(new InetSocketAddress(InetAddress.*getByName*("IP"),port));  
acceptorSvr.configureBlocking(false);

步骤三：创建Reactor线程，创建多路复用器并启动线程。

Selector selector = Selector.*open*();  
new Thread(new ReactorTask()).start();

步骤四：将ServerSocketChannel注册到Reactor线程的多路复用器Selector上，监听Accept事件，

acceptorSvr.register(selector, SelectionKey.*OP\_ACCEPT*,ioHandler);

步骤五：多路复用器在线程run方法的无限循环体内轮询准备就绪的Key，

int num = selector.select();  
Set<SelectionKey> selectionKeys = selector.selectedKeys();  
Iterator it = selectionKeys.iterator();  
while (it.hasNext()){  
 SelectionKey key = (SelectionKey) it.next();  
 // ... deal with I/O event ...  
}

步骤六：多路复用器监听到有新的客户端接入，处理新的接入请求，完成TCP三次握手，建立物理链路，实例代码如下：

SocketChannel channel = acceptorSvr.accept();

步骤七：设置客户端链路为非阻塞模式，

channel.configureBlocking(false);  
channel.socket().setReuseAddress(true);

步骤八：将新计入的客户端连接注册到Reactor线程的多路复用器上，监听读操作，读取客户端发送的网路消息，

SelectionKey key = channel.register(selector, SelectionKey.*OP\_READ*, ioHandler);

步骤九：异步读取客户端请求消息到换从区，

int readNumber = channel.read(receivedBuffer);

步骤十：对ByteBuffer进行彪马，如果有半包消息指针reset，继续读取后续的报文，将解码成功的消息封装成Task，投递到业务线程中，进行业务逻辑编排，实例代码如下

Object message = null;  
while (byteBuffer.hasRemaining()) {  
 byteBuffer.mark();  
 message = *decode*(byteBuffer);  
 if (message == null) {  
 byteBuffer.reset();  
 break;  
 }  
 messageList.add(message);  
  
}  
  
if(!byteBuffer.hasRemaining()){  
 byteBuffer.clear();  
}else {  
 byteBuffer.compact();  
}  
  
if(messageList != null && !messageList.isEmpty()){  
 for (Object messageE : messageList) {  
 *handlerTask*(messageE);  
 }  
}

步骤十一：将POJO对象encode成ByteBuffer没调用SocketChannel的异步write接口将消息异步发送给客户端

channel.write(byteBuffer);

### 2.3.3 创建TimeServer2

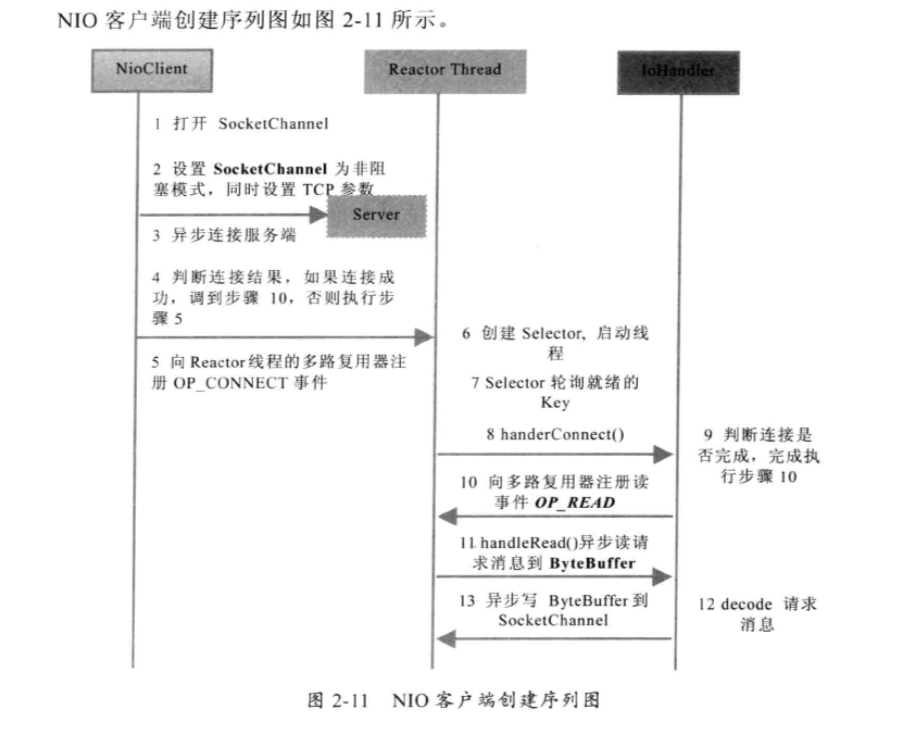
TimeServer2

|  |
| --- |
| package com.netty.study.chapter02;  import java.io.IOException;  public class TimeServer2 {   public static void main(String[] args) throws IOException {   int port = 8080;  if (args != null && args.length > 0) {  try {  port = Integer.*valueOf*(args[0]);  } catch (NumberFormatException e) {  }  }   MultiplexerTimeServer timeServer = new MultiplexerTimeServer(port);   new Thread(timeServer,"NIO-MultiplexerTimeServer-001").start();  } } |

MultiplexerTimeServer

|  |
| --- |
| package com.netty.study.chapter02;  import java.io.IOException; import java.net.InetSocketAddress; import java.nio.ByteBuffer; import java.nio.channels.SelectionKey; import java.nio.channels.Selector; import java.nio.channels.ServerSocketChannel; import java.nio.channels.SocketChannel; import java.util.Date; import java.util.Iterator; import java.util.Set;  public class MultiplexerTimeServer implements Runnable {   private Selector selector;   private ServerSocketChannel servChannel;   private volatile boolean stop;   */\*\*  \* 初始化多路复用器，绑定监听端口  \*  \** ***@param*** *port  \*/* public MultiplexerTimeServer(int port) {  try {  selector = Selector.*open*();  servChannel = ServerSocketChannel.*open*();  servChannel.configureBlocking(false);  servChannel.socket().bind(new InetSocketAddress(port), 1024);  servChannel.register(selector, SelectionKey.*OP\_ACCEPT*);  System.*out*.println("The time server is start in port :" + port);   } catch (IOException e) {  e.printStackTrace();  System.*exit*(1);  }  }   public void stop() {  this.stop = true;  }    public void run() {  while (!stop) {  try {  selector.select(1000);  Set<SelectionKey> selectionKeys = selector.selectedKeys();  Iterator<SelectionKey> it = selectionKeys.iterator();   SelectionKey key = null;  while (it.hasNext()) {  key = it.next();  it.remove();  try {  handleInput(key);  } catch (Exception e) {  if (key != null) {  key.cancel();  if (key.channel() != null) {  key.channel().close();  }  }  }  }   } catch (Throwable t) {  t.printStackTrace();  }  }   // 多路复用器关闭后，所有注册在上面的Channel和Pipe等资源会被自动去注册并关闭，所以不需要重复释放资源  if (selector != null) {  try {  selector.close();  } catch (IOException e) {  e.printStackTrace();  }  }  }   private void handleInput(SelectionKey key) throws IOException {  if (key.isValid()) {  if (key.isAcceptable()) {  //处理新接入的请求信息  ServerSocketChannel ssc = (ServerSocketChannel) key.channel();  SocketChannel sc = ssc.accept();  sc.configureBlocking(false);   sc.register(selector, SelectionKey.*OP\_READ*);  }  if (key.isReadable()) {  // Read the data  SocketChannel sc = (SocketChannel) key.channel();  ByteBuffer readBuffer = ByteBuffer.*allocate*(1024);  int readBytes = sc.read(readBuffer);   if (readBytes > 0) {  readBuffer.flip();  byte[] bytes = new byte[readBuffer.remaining()];  readBuffer.get(bytes);   String body = new String(bytes, "UTF-8");  System.*out*.println("The time server receive order: " + body);  String currentTime = "QUERY TIME ORDER".equalsIgnoreCase(body) ? new Date(System.*currentTimeMillis*()).toString() : "BAD ORDER";  doWrite(sc, currentTime);  } else if (readBytes < 0) {  // 对端链路关闭  key.cancel();  sc.close();  } else {  // 读到0字节，忽略  }  }  }    }   private void doWrite(SocketChannel channel, String response) throws IOException {  if (response != null && response.trim().length() > 0) {  byte[] bytes = response.getBytes();  ByteBuffer writeBuffer = ByteBuffer.*allocate*(bytes.length);  writeBuffer.put(bytes);  writeBuffer.flip();  channel.write(writeBuffer);  }  } } |

### 2.3.4 客户端序列图



### 2.3.5 NIO创建TimeClient

TimeCilent

|  |
| --- |
| package com.netty.study.chapter02;  import java.io.BufferedReader; import java.io.IOException; import java.io.InputStreamReader; import java.io.PrintWriter; import java.net.Socket;  public class TimeClient2 {   public static void main(String[] args) {  int port = 8080;  if (args != null && args.length > 0) {  try {  port = Integer.*valueOf*(args[0]);  } catch (NumberFormatException e) {   }  }   new Thread(new TimeClientHandle("127.0.0.1",port),"TimeClient-001").start();   }  } |

TimeClientHandle

|  |
| --- |
| package com.netty.study.chapter02;  import java.io.IOException; import java.net.InetSocketAddress; import java.nio.ByteBuffer; import java.nio.channels.SelectionKey; import java.nio.channels.Selector; import java.nio.channels.SocketChannel; import java.util.Iterator; import java.util.Set;  public class TimeClientHandle implements Runnable {   private String host;  private int port;  private Selector selector;  private SocketChannel socketChannel;  private volatile boolean stop;    public TimeClientHandle(String host, int port) {  this.host = host == null ? "127.0.0.1" : host;  this.port = port;   try {  selector = Selector.*open*();  socketChannel = SocketChannel.*open*();  socketChannel.configureBlocking(false);  } catch (IOException e) {  e.printStackTrace();  System.*exit*(1);  }  }   public void run() {  try {  doConnect();  } catch (IOException e) {  e.printStackTrace();  System.*exit*(1);  }   while (!stop) {  try {  selector.select(1000);  Set<SelectionKey> selectionKeys = selector.selectedKeys();  Iterator<SelectionKey> it = selectionKeys.iterator();  SelectionKey key = null;   while (it.hasNext()) {  key = it.next();  it.remove();  try {  handleInput(key);  } catch (Exception e) {  if (key != null) {  key.cancel();  if (key.channel() != null) {  key.channel().close();  }  }  }   }  } catch (Exception e) {  e.printStackTrace();  System.*exit*(1);  }  }   // 多路复用器关闭后，所有注册在上面的Channel和Pipe等资源都会被自动去注册并关闭，所以不需要重复释放资源  if (selector != null) {  try {  selector.close();  } catch (IOException e) {  e.printStackTrace();  }  }    }   private void handleInput(SelectionKey key) throws IOException {  if (key.isValid()) {  // 判断是否连接成功  SocketChannel sc = (SocketChannel) key.channel();  if (key.isConnectable()) {  if (sc.finishConnect()) {  sc.register(selector, SelectionKey.*OP\_READ*);  doWrite(sc);  } else {  // 连接失败，进程退出  System.*exit*(1);  }  }   if (key.isReadable()) {  ByteBuffer readBuffer = ByteBuffer.*allocate*(1024);  int readBytes = sc.read(readBuffer);  if (readBytes > 0) {  readBuffer.flip();  byte[] bytes = new byte[readBuffer.remaining()];  readBuffer.get(bytes);  String body = new String(bytes, "UTF-8");  System.*out*.println("Now is : " + body);  this.stop = true;  } else if (readBytes < 0) {  key.cancel();  sc.close();  } else {  // 读到0字节，忽略  }  }  }    }   private void doWrite(SocketChannel sc) throws IOException {  byte[] req = "QUERY TIME ORDER".getBytes();  ByteBuffer writeBuffer = ByteBuffer.*allocate*(req.length);  writeBuffer.put(req);  writeBuffer.flip();  sc.write(writeBuffer);  if (!writeBuffer.hasRemaining()) {  System.*out*.println("Send order 2 server succeed.");  }   }   private void doConnect() throws IOException {  if (socketChannel.connect(new InetSocketAddress(host, port))) {  socketChannel.register(selector, SelectionKey.*OP\_READ*);  doWrite(socketChannel);  } else {  socketChannel.register(selector, SelectionKey.*OP\_CONNECT*);  }  }   } |

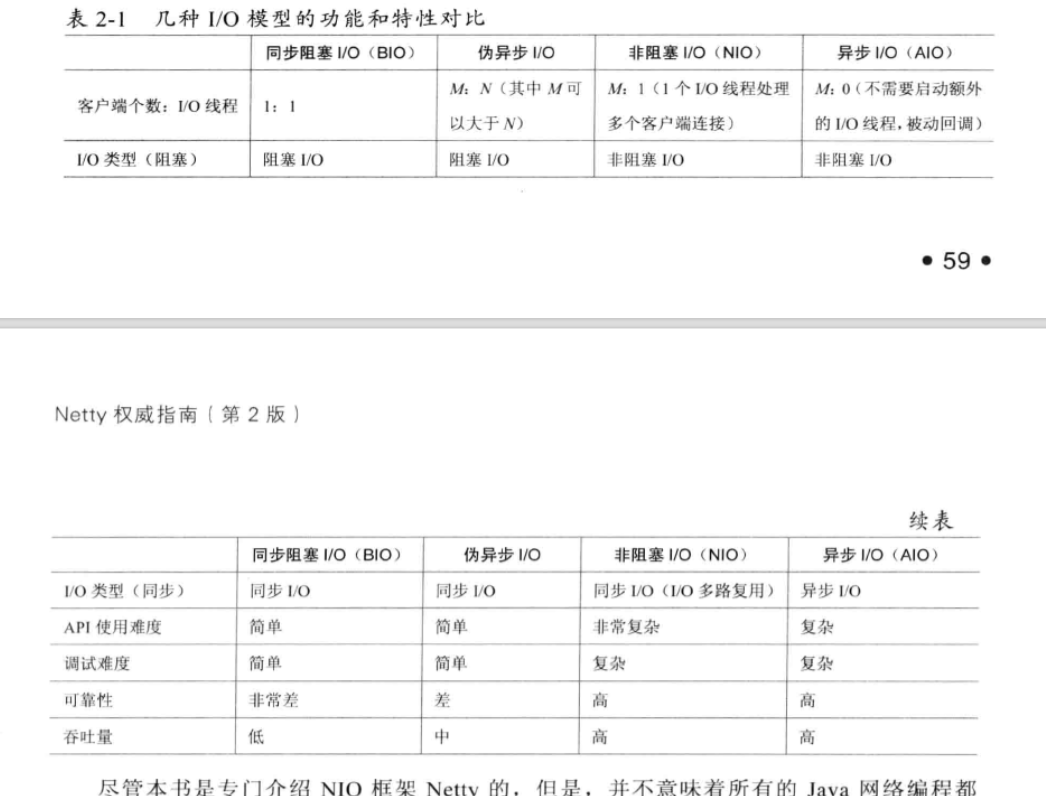
## 2.4 AIO编程

|  |
| --- |
| package com.netty.study.chapter02;  import java.io.IOException;  public class TimeServer3 {   public static void main(String[] args) throws IOException {   int port = 8080;  if (args != null && args.length > 0) {  try {  port = Integer.*valueOf*(args[0]);  } catch (NumberFormatException e) {  }  }   AsyncTimeServerHandler timeServer = new AsyncTimeServerHandler(port);   new Thread(timeServer,"NIO-AsyncTimeServerHandler-001").start();  } }  package com.netty.study.chapter02;  import java.io.IOException; import java.net.InetSocketAddress; import java.nio.channels.AsynchronousServerSocketChannel; import java.util.concurrent.CountDownLatch;  public class AsyncTimeServerHandler implements Runnable {   private int port;  CountDownLatch latch;  AsynchronousServerSocketChannel asynchronousServerSocketChannel;   public AsyncTimeServerHandler(int port) {  this.port = port;   try {  asynchronousServerSocketChannel = AsynchronousServerSocketChannel.*open*();  asynchronousServerSocketChannel.bind(new InetSocketAddress(port));  System.*out*.println("The time server is start in port : " + port);   } catch (IOException e) {  e.printStackTrace();  }   }   public void run() {   latch = new CountDownLatch(1);  doAccept();  try {  latch.await();  }catch (InterruptedException e){  e.printStackTrace();  }   }   private void doAccept() {  asynchronousServerSocketChannel.accept(this,new AcceptCompletionHandler());  } }  package com.netty.study.chapter02;  import java.nio.ByteBuffer; import java.nio.channels.AsynchronousSocketChannel; import java.nio.channels.CompletionHandler;  public class AcceptCompletionHandler implements CompletionHandler<AsynchronousSocketChannel, AsyncTimeServerHandler> {   @Override  public void completed(AsynchronousSocketChannel result, AsyncTimeServerHandler attachment) {  attachment.asynchronousServerSocketChannel.accept(attachment, this);  ByteBuffer buffer = ByteBuffer.*allocate*(1024);  result.read(buffer, buffer, new ReadCompletionHandler(result));  }   @Override  public void failed(Throwable exc, AsyncTimeServerHandler attachment) {  exc.printStackTrace();  attachment.latch.countDown();  } }  package com.netty.study.chapter02;  import java.io.IOException; import java.io.UnsupportedEncodingException; import java.nio.ByteBuffer; import java.nio.channels.AsynchronousSocketChannel; import java.nio.channels.CompletionHandler; import java.util.Date;  public class ReadCompletionHandler implements CompletionHandler<Integer, ByteBuffer> {   private AsynchronousSocketChannel channel;   public ReadCompletionHandler(AsynchronousSocketChannel channel) {  if (this.channel == null) {  this.channel = channel;  }   }   @Override  public void completed(Integer result, ByteBuffer attachment) {  attachment.flip();  byte[] body = new byte[attachment.remaining()];  attachment.get(body);   try {  String req = new String(body, "UTF-8");  System.*out*.println("The time server receive order : " + req);  String currentTime = "QUERY TIME ORDER".equalsIgnoreCase(req) ? new Date(System.*currentTimeMillis*()).toString() : "BAD ORDER";  doWrite(currentTime);  } catch (UnsupportedEncodingException e) {  e.printStackTrace();  }  }   private void doWrite(String currentTime) {  if (currentTime != null && currentTime.trim().length() > 0) {  byte[] bytes = (currentTime).getBytes();  final ByteBuffer writeBuffer = ByteBuffer.*allocate*(bytes.length);  writeBuffer.put(bytes);  writeBuffer.flip();  channel.write(writeBuffer, writeBuffer, new CompletionHandler<Integer, ByteBuffer>() {  @Override  public void completed(Integer result, ByteBuffer buffer) {  // 如果没有发送完成，继续发送  if (buffer.hasRemaining()) {  channel.write(buffer, buffer, this);  }  }   @Override  public void failed(Throwable exc, ByteBuffer attachment) {  try {  channel.close();  } catch (IOException e) {  //ingnore on close  }  }  });  }  }   @Override  public void failed(Throwable exc, ByteBuffer attachment) {  try {  this.channel.close();  } catch (IOException e) {  e.printStackTrace();  }  } } |

客户端

|  |
| --- |
| package com.netty.study.chapter02;  public class TimeClient3 {   public static void main(String[] args) {  int port = 8080;  if (args != null && args.length > 0) {  try {  port = Integer.*valueOf*(args[0]);  } catch (NumberFormatException e) {   }  }  new Thread(new AsyncTimeClientHandler("127.0.0.1", port), "AsyncTimeClientHandler-001").start();  }  }  package com.netty.study.chapter02;  import java.io.IOException; import java.io.UnsupportedEncodingException; import java.net.InetSocketAddress; import java.nio.ByteBuffer; import java.nio.channels.AsynchronousSocketChannel; import java.nio.channels.CompletionHandler; import java.util.concurrent.CountDownLatch;  public class AsyncTimeClientHandler implements CompletionHandler<Void, AsyncTimeClientHandler>, Runnable {   private AsynchronousSocketChannel client;  private String host;  private int port;  private CountDownLatch latch;    public AsyncTimeClientHandler(String host, int port) {  this.host = host;  this.port = port;   try {  client = AsynchronousSocketChannel.*open*();  } catch (IOException e) {  e.printStackTrace();  }   }   @Override  public void run() {  latch = new CountDownLatch(1);  client.connect(new InetSocketAddress(host, port), this, this);   try {  latch.await();  } catch (InterruptedException e1) {  e1.printStackTrace();  }   try {  client.close();  } catch (IOException e) {  e.printStackTrace();  }   }   @Override  public void completed(Void result, AsyncTimeClientHandler attachment) {  byte[] req = "QUERY TIME ORDER".getBytes();  ByteBuffer writeBuffer = ByteBuffer.*allocate*(req.length);  writeBuffer.put(req);  writeBuffer.flip();   client.write(writeBuffer, writeBuffer, new CompletionHandler<Integer, ByteBuffer>() {  @Override  public void completed(Integer result, ByteBuffer buffer) {  if (buffer.hasRemaining()) {  client.write(buffer, buffer, this);  } else {  ByteBuffer readBuffer = ByteBuffer.*allocate*(1024);  client.read(readBuffer, readBuffer, new CompletionHandler<Integer, ByteBuffer>() {  @Override  public void completed(Integer result, ByteBuffer buffer) {  buffer.flip();  byte[] bytes = new byte[buffer.remaining()];  buffer.get(bytes);  String body;  try {  body = new String(bytes, "UTF-8");  System.*out*.println("Now is:" + body);  } catch (UnsupportedEncodingException e) {  e.printStackTrace();  }  }   @Override  public void failed(Throwable exc, ByteBuffer attachment) {  try {  client.close();  latch.countDown();  } catch (IOException e) {  e.printStackTrace();  // ingnore on close  }  }  });  }  }   @Override  public void failed(Throwable exc, ByteBuffer attachment) {  try {  client.close();  latch.countDown();  } catch (IOException e) {  e.printStackTrace();  }  }  });  }    @Override  public void failed(Throwable exc, AsyncTimeClientHandler attachment) {  try {  client.close();  latch.countDown();  } catch (IOException e) {  e.printStackTrace();  }  } } |

## 2.5 4中IO对比



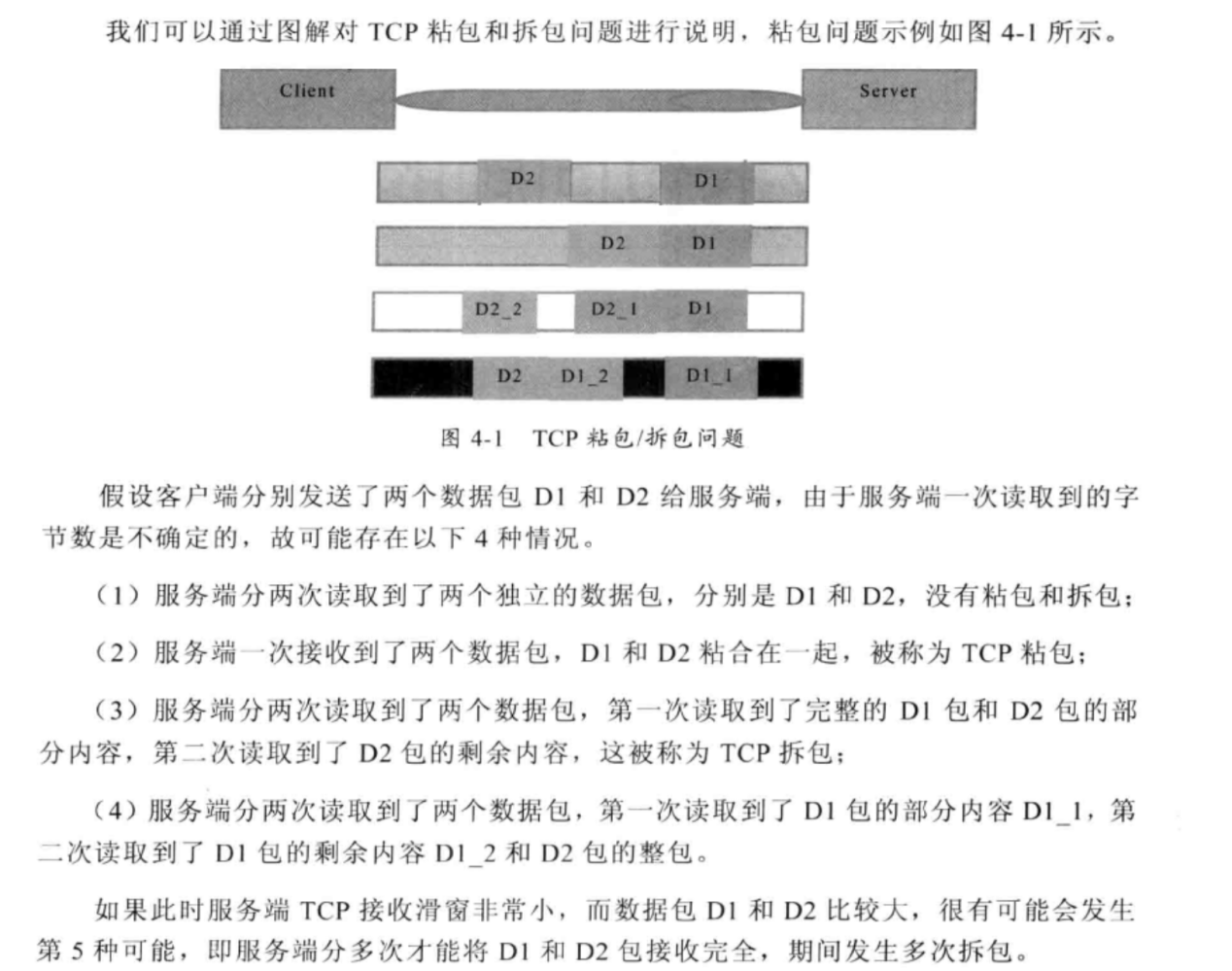
## 2.6 选择netty理由

# Netty入门的搭建

# TCP粘包/拆包问题解决

## 4.1 TCP粘包/拆包

### 4.1.1 TCP粘包/拆包问题说明



### 4.1.2 粘包/拆包发生的原因

1. 应用程序write写入的字节大小小于套接口发送缓存区大小；
2. 进行MSS大小的TCP分段；
3. 以太网帧的payload大于MTU进行IP分片



### 4.1.3 粘包问题的解决策略

1. 消息定长，例如每个报文的大小为固定长度200字节，如果不够，空位补空格
2. 在包尾增加回车换行符进行分割，例如FTP协议
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
4. 更为复杂的应用层协议

## 4.2 未考虑TCP粘包/拆包导致功能异常案例

Server端

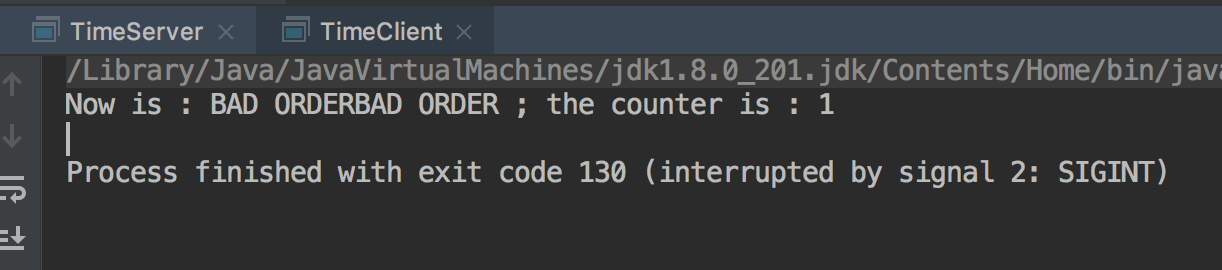
|  |
| --- |
| package com.netty.study.chapter04;  import io.netty.bootstrap.ServerBootstrap; import io.netty.channel.ChannelFuture; import io.netty.channel.ChannelInitializer; import io.netty.channel.ChannelOption; import io.netty.channel.nio.NioEventLoopGroup; import io.netty.channel.socket.SocketChannel; import io.netty.channel.socket.nio.NioServerSocketChannel;  public class TimeServer {   public void bind(int port) throws Exception {  // 配置服务端的NIO线程组  NioEventLoopGroup bossGroup = new NioEventLoopGroup();  NioEventLoopGroup workerGroup = new NioEventLoopGroup();   try {  ServerBootstrap b = new ServerBootstrap();  b.group(bossGroup, workerGroup)  .channel(NioServerSocketChannel.class)  .option(ChannelOption.*SO\_BACKLOG*, 1024)  .childHandler(new TimeServer.ChildChannelHandler());  // 绑定端口，同步等待成功  ChannelFuture f = b.bind(port).sync();   // 等待服务端监听端口关闭  f.channel().closeFuture().sync();   } finally {  // 优雅退出，释放线程池资源  bossGroup.shutdownGracefully();  workerGroup.shutdownGracefully();  }  }      private class ChildChannelHandler extends ChannelInitializer<SocketChannel> {   @Override  protected void initChannel(SocketChannel socketChannel) throws Exception {  socketChannel.pipeline().addLast(new TimeServerHandler());  }  }    public static void main(String[] args) throws Exception {  new com.netty.study.chapter03.TimeServer().bind(8080);  }    }  package com.netty.study.chapter04;  import io.netty.buffer.ByteBuf; import io.netty.buffer.Unpooled; import io.netty.channel.ChannelHandlerAdapter; import io.netty.channel.ChannelHandlerContext;  import java.util.Date;  public class TimeServerHandler extends ChannelHandlerAdapter {   private int counter;   @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").substring(0, req.length - System.*getProperty*("line.separator").length());  System.*out*.println("The time server receive order : " + body + " ; the counter is : " + ++counter);  String currentTime = "QUERY TIME ORDER".equalsIgnoreCase(body) ? new Date(System.*currentTimeMillis*()).toString() : "BAD ORDER";  currentTime = currentTime + System.*getProperty*("line.separator");   ByteBuf resp = Unpooled.*copiedBuffer*(currentTime.getBytes());  ctx.writeAndFlush(resp);   }    @Override  public void channelReadComplete(ChannelHandlerContext ctx) throws Exception {  ctx.flush();  }   @Override  public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws Exception {  ctx.close();  } } |

client端

|  |
| --- |
| package com.netty.study.chapter04;  import io.netty.bootstrap.Bootstrap; 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.NioSocketChannel;  public class TimeClient {   public void connect(int port, String host) throws Exception {  // 配置客户端NIO线程组  EventLoopGroup group = new NioEventLoopGroup();   try {  Bootstrap b = new Bootstrap();  b.group(group).channel(NioSocketChannel.class)  .option(ChannelOption.*TCP\_NODELAY*, true)  .handler(new ChannelInitializer<SocketChannel>() {  @Override  protected void initChannel(SocketChannel socketChannel) throws Exception {  socketChannel.pipeline().addLast(new TimeClietHandler());  }  });  // 发起异步连接操作  ChannelFuture f = b.connect(host, port).sync();  // 等待客户独断链路操作  f.channel().closeFuture().sync();  } finally {  group.shutdownGracefully();  }  }   public static void main(String[] args) throws Exception {  new TimeClient().connect(8080,"127.0.0.1");  }  }  package com.netty.study.chapter04;  import io.netty.buffer.ByteBuf; import io.netty.buffer.Unpooled; import io.netty.channel.ChannelHandlerAdapter; import io.netty.channel.ChannelHandlerContext;  public class TimeClientHandler extends ChannelHandlerAdapter {    private int counter;   private byte[] req;    public TimeClientHandler() {  req = ("QUERY TIME ORDER" + System.*getProperty*("line.separator")).getBytes();   }    @Override  public void channelActive(ChannelHandlerContext ctx) throws Exception {  ByteBuf message = null;  // 客户端跟服务端链路建立成功之后，循环发送100条消息，没发送一条就刷新一次，  // 保证每条消息都会被写入Channel中  for (int i = 0; i < 100; i++) {  message = Unpooled.*buffer*(req.length);  message.writeBytes(req);  ctx.writeAndFlush(message);  }  }   @Override  public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {  // 客户端每接收服务端一条消息应答之后，就打印一次计数器。  ByteBuf buf = (ByteBuf) msg;  byte[] req = new byte[buf.readableBytes()];  buf.readBytes(req);  String body = new String(req, "UTF-8");  System.*out*.println("Now is : " + body + " ; the counter is : " + (++counter));  }   @Override  public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws Exception {  // 释放资源  System.*out*.println("Unexpected exception from downstream : " + cause.getMessage());  ctx.close();  } } |

运行结果是：服务端接收两条消息，客户端接收一条消息

客户端，接收到含有2个 BAD ORDER的一条消息



服务端接收2条消息：第二条和第一条 消息 发生了部分粘包，消息 被分开



## 4.3 利用LineBasedFrameDecoder解决粘包问题

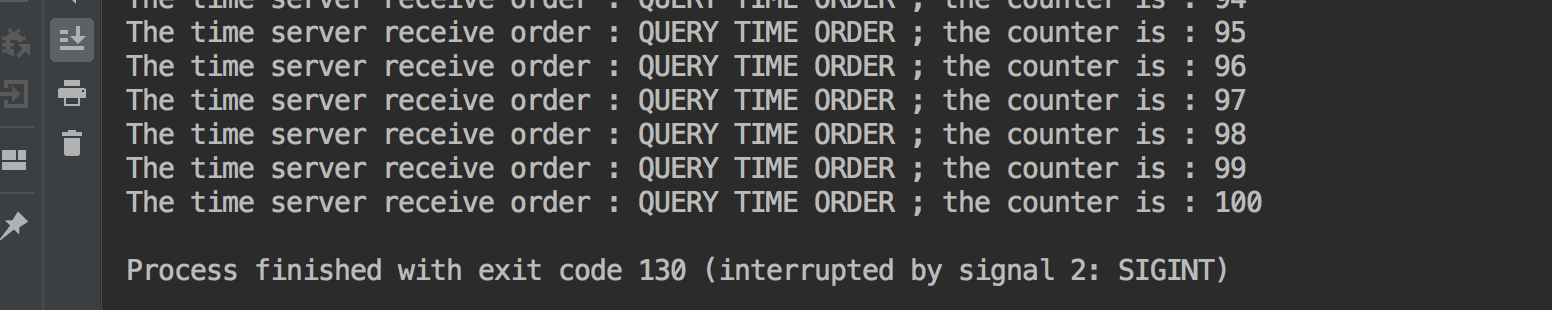
Server端

|  |
| --- |
| package com.netty.study.chapter04;  import io.netty.bootstrap.ServerBootstrap; import io.netty.channel.ChannelFuture; import io.netty.channel.ChannelInitializer; import io.netty.channel.ChannelOption; import io.netty.channel.nio.NioEventLoopGroup; import io.netty.channel.socket.SocketChannel; import io.netty.channel.socket.nio.NioServerSocketChannel; import io.netty.handler.codec.LineBasedFrameDecoder; import io.netty.handler.codec.string.StringDecoder;  public class TimeServer2 {   public void bind(int port) throws Exception {  // 配置服务端的NIO线程组  NioEventLoopGroup bossGroup = new NioEventLoopGroup();  NioEventLoopGroup workerGroup = new NioEventLoopGroup();   try {  ServerBootstrap b = new ServerBootstrap();  b.group(bossGroup, workerGroup)  .channel(NioServerSocketChannel.class)  .option(ChannelOption.*SO\_BACKLOG*, 1024)  .childHandler(new ChildChannelHandler());  // 绑定端口，同步等待成功  ChannelFuture f = b.bind(port).sync();   // 等待服务端监听端口关闭  f.channel().closeFuture().sync();   } finally {  // 优雅退出，释放线程池资源  bossGroup.shutdownGracefully();  workerGroup.shutdownGracefully();  }  }    private class ChildChannelHandler extends ChannelInitializer<SocketChannel> {   @Override  protected void initChannel(SocketChannel socketChannel) throws Exception {  socketChannel.pipeline().addLast(new LineBasedFrameDecoder(1024));  socketChannel.pipeline().addLast(new StringDecoder());  socketChannel.pipeline().addLast(new TimeServerHandler2());  }  }    public static void main(String[] args) throws Exception {  new TimeServer2().bind(8080);  }   }  package com.netty.study.chapter04;  import io.netty.buffer.ByteBuf; import io.netty.buffer.Unpooled; import io.netty.channel.ChannelHandlerAdapter; import io.netty.channel.ChannelHandlerContext;  import java.util.Date;  public class TimeServerHandler2 extends ChannelHandlerAdapter {   private int counter;   @Override  public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {   String body = (String) msg;  System.*out*.println("The time server receive order : " + body + " ; the counter is : " + ++counter);  String currentTime = "QUERY TIME ORDER".equalsIgnoreCase(body) ? new Date(System.*currentTimeMillis*()).toString() : "BAD ORDER";  currentTime = currentTime + System.*getProperty*("line.separator");   ByteBuf resp = Unpooled.*copiedBuffer*(currentTime.getBytes());  ctx.writeAndFlush(resp);   }   // @Override // public void channelReadComplete(ChannelHandlerContext ctx) throws Exception { // ctx.flush(); // }   @Override  public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws Exception {  ctx.close();  } } |

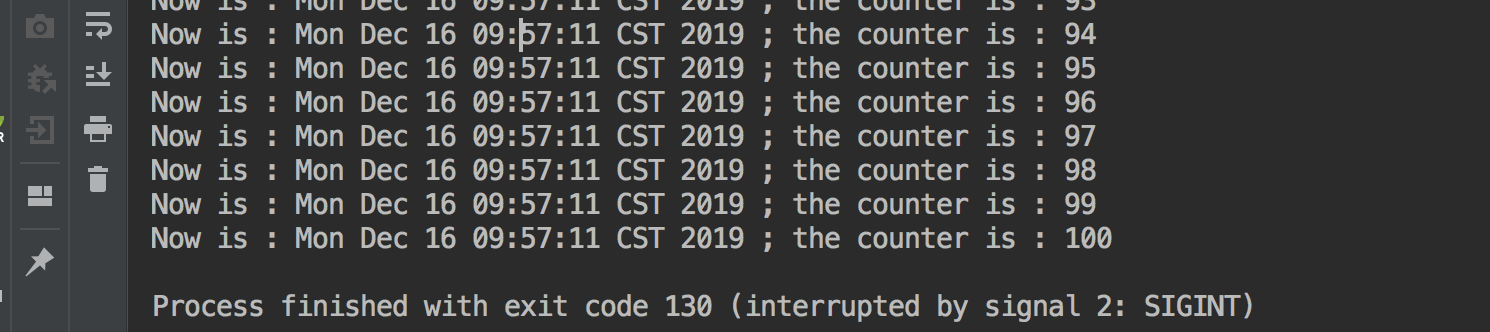
client

|  |
| --- |
| package com.netty.study.chapter04;  import io.netty.bootstrap.Bootstrap; 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.NioSocketChannel; import io.netty.handler.codec.LineBasedFrameDecoder; import io.netty.handler.codec.string.StringDecoder;  public class TimeClient2 {   public void connect(int port, String host) throws Exception {  // 配置客户端NIO线程组  EventLoopGroup group = new NioEventLoopGroup();   try {  Bootstrap b = new Bootstrap();  b.group(group).channel(NioSocketChannel.class)  .option(ChannelOption.*TCP\_NODELAY*, true)  .handler(new ChannelInitializer<SocketChannel>() {  @Override  protected void initChannel(SocketChannel socketChannel) throws Exception {  socketChannel.pipeline().addLast(new LineBasedFrameDecoder(1024));  socketChannel.pipeline().addLast(new StringDecoder());  socketChannel.pipeline().addLast(new TimeClientHandler2());   }  });  // 发起异步连接操作  ChannelFuture f = b.connect(host, port).sync();  // 等待客户独断链路操作  f.channel().closeFuture().sync();  } finally {  group.shutdownGracefully();  }  }   public static void main(String[] args) throws Exception {  new TimeClient2().connect(8080, "127.0.0.1");  }  }  package com.netty.study.chapter04;  import io.netty.buffer.ByteBuf; import io.netty.buffer.Unpooled; import io.netty.channel.ChannelHandlerAdapter; import io.netty.channel.ChannelHandlerContext;  public class TimeClientHandler2 extends ChannelHandlerAdapter {    private int counter;   private byte[] req;    public TimeClientHandler2() {  req = ("QUERY TIME ORDER" + System.*getProperty*("line.separator")).getBytes();   }    @Override  public void channelActive(ChannelHandlerContext ctx) throws Exception {  ByteBuf message = null;  // 客户端跟服务端链路建立成功之后，循环发送100条消息，没发送一条就刷新一次，  // 保证每条消息都会被写入Channel中  for (int i = 0; i < 100; i++) {  message = Unpooled.*buffer*(req.length);  message.writeBytes(req);  ctx.writeAndFlush(message);  }  }   @Override  public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {  // 客户端每接收服务端一条消息应答之后，就打印一次计数器。  String body = (String) msg;  System.*out*.println("Now is : " + body + " ; the counter is : " + (++counter));  }   @Override  public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws Exception {  // 释放资源  System.*out*.println("Unexpected exception from downstream : " + cause.getMessage());  ctx.close();  } } |

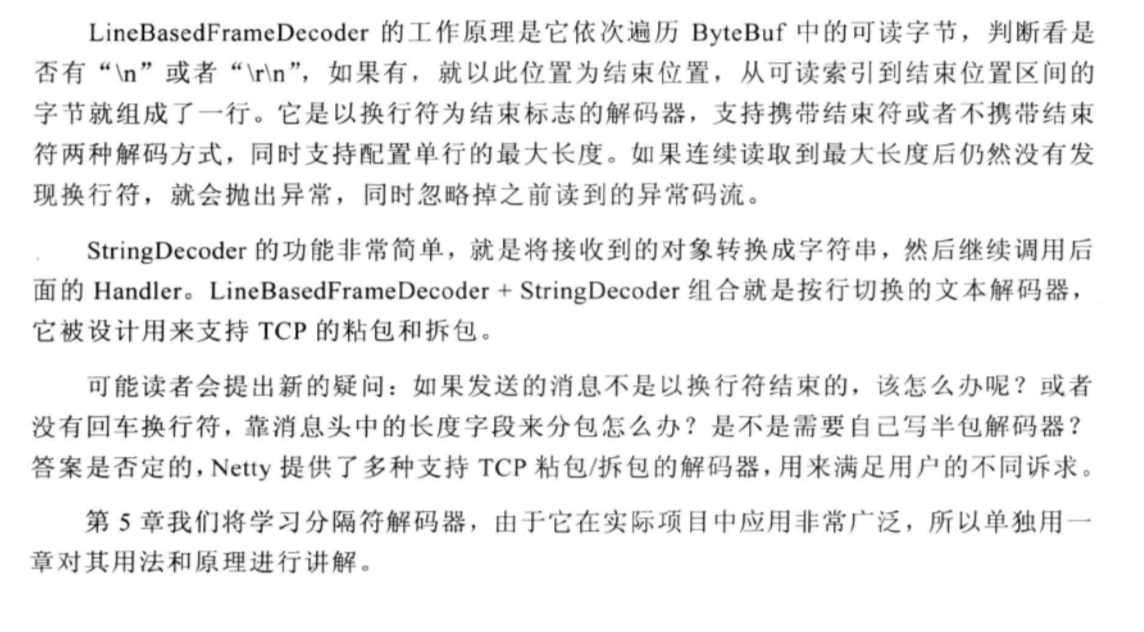
server端输出：



client输出：



### 4.3.4 LineBasedFrameDecoder和StringDecoder的原理



# 分隔符和定长解码器的应用

上层的应用协议为了对消息进行区分，往往采用如下4种方式。

1. 消息长度固定，累计读取到长度总和为定长LEN的报文后，就认为读取到了一个完整的消息；将计数器置位，重新开始读取下一个数据报。
2. 将回车换行符作为消息结束符，例如FTP协议，这种方式在文本协议中应用比较广泛
3. 将特殊的分隔符作为消息的结束标志，回车换行符就是一种特殊的结束分隔符
4. 通过在消息头中定义长度字段来表示消息的总长度。

## 5.1 DelimiterBasedFrameDecoder服务端和客户端

server

|  |
| --- |
| package com.netty.study.chapter05;  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.string.StringDecoder; import io.netty.handler.logging.LogLevel; import io.netty.handler.logging.LoggingHandler;  public class EchoServer {   public void bind(int port) throws Exception {  // 配置服务端的NIO线程组  EventLoopGroup bossGroup = new NioEventLoopGroup();  EventLoopGroup workerGroup = new NioEventLoopGroup();   try {  ServerBootstrap b = new ServerBootstrap();  b.group(bossGroup, workerGroup)  .channel(NioServerSocketChannel.class)  .option(ChannelOption.*SO\_BACKLOG*, 100)  .handler(new LoggingHandler(LogLevel.*INFO*))  .childHandler(new ChannelInitializer<SocketChannel>() {  @Override  protected void initChannel(SocketChannel socketChannel) throws Exception {  ByteBuf delimiter = Unpooled.*copiedBuffer*("$\_".getBytes());  socketChannel.pipeline().addLast(new DelimiterBasedFrameDecoder(1024, delimiter));  socketChannel.pipeline().addLast(new StringDecoder());  socketChannel.pipeline().addLast(new EchoServerHandler());  }  });  // 绑定端口，同步等待成功  ChannelFuture f = b.bind(port).sync();   // 等待服务端监听端口关闭  f.channel().closeFuture().sync();  } finally {  bossGroup.shutdownGracefully();  workerGroup.shutdownGracefully();  }  }   public static void main(String[] args) throws Exception {  new EchoServer().bind(8080);  }   }  package com.netty.study.chapter05;  import io.netty.buffer.ByteBuf; import io.netty.buffer.Unpooled; import io.netty.channel.ChannelHandler; import io.netty.channel.ChannelHandlerAdapter; import io.netty.channel.ChannelHandlerContext; import io.netty.channel.ChannelHandlerInvoker; import io.netty.util.concurrent.EventExecutorGroup;  public class EchoServerHandler extends ChannelHandlerAdapter {   int counter = 0;   @Override  public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {  String body = (String) msg;  System.*out*.println("This is " + (++counter) + " times receive client: [" + body + "]");  body += "$\_";  ByteBuf echo = Unpooled.*copiedBuffer*(body.getBytes());  ctx.writeAndFlush(echo);  }   @Override  public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws Exception {  cause.printStackTrace();  ctx.close();// 发生异常，关闭链路  } } |

client

|  |
| --- |
| package com.netty.study.chapter05;  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.ChannelOption; 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.string.StringDecoder;  public class EchoClient {    public void connect(int port, String host) throws Exception {  // 配置客户端NIO线程组  EventLoopGroup group = new NioEventLoopGroup();  try {  Bootstrap b = new Bootstrap();  b.group(group).channel(NioSocketChannel.class)  .option(ChannelOption.*TCP\_NODELAY*,true)  .handler(new ChannelInitializer<SocketChannel>() {  @Override  protected void initChannel(SocketChannel socketChannel) throws Exception {  ByteBuf delimiter = Unpooled.*copiedBuffer*("$\_".getBytes());  socketChannel.pipeline().addLast(new DelimiterBasedFrameDecoder(1024,delimiter));  socketChannel.pipeline().addLast(new StringDecoder());  socketChannel.pipeline().addLast(new EchoClientHandler());  }  });  // 发起异步连接操作  ChannelFuture f = b.connect(host,port).sync();   // 等待客户端链路关闭  f.channel().closeFuture().sync();  }finally {  // 优雅突出，释放NIO线程组  group.shutdownGracefully();  }  }   public static void main(String[] args) throws Exception {  new EchoClient().connect(8080,"127.0.0.1");  }  }  package com.netty.study.chapter05;  import io.netty.buffer.Unpooled; import io.netty.channel.ChannelHandlerAdapter; import io.netty.channel.ChannelHandlerContext;  public class EchoClientHandler extends ChannelHandlerAdapter {   private int counter;   static final String *ECHO\_REQ* = "Hi,Lilinfeng. Welcome to Netty.$\_";   public EchoClientHandler() {  }   @Override  public void channelActive(ChannelHandlerContext ctx) throws Exception {  for (int i = 0; i < 10; i++) {  ctx.writeAndFlush(Unpooled.*copiedBuffer*(*ECHO\_REQ*.getBytes()));  }  }   @Override  public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {  System.*out*.println("This is " + (++counter) + " times receive server : " + msg);  }   @Override  public void channelReadComplete(ChannelHandlerContext ctx) throws Exception {  ctx.flush();  }   @Override  public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws Exception {  cause.printStackTrace();  ctx.close();  } } |

## 5.2 FixedLengthFrameDecoder应用开发

server

|  |
| --- |
| package com.netty.study.chapter05;  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.logging.LogLevel; import io.netty.handler.logging.LoggingHandler;  public class EchoServer2 {   public void bind(int port) throws Exception {  // 配置服务端的NIO线程组  EventLoopGroup bossGroup = new NioEventLoopGroup();  EventLoopGroup workerGroup = new NioEventLoopGroup();   try {  ServerBootstrap b = new ServerBootstrap();  b.group(bossGroup, workerGroup)  .channel(NioServerSocketChannel.class)  .option(ChannelOption.*SO\_BACKLOG*, 100)  .handler(new LoggingHandler(LogLevel.*INFO*))  .childHandler(new ChannelInitializer<SocketChannel>() {  @Override  protected void initChannel(SocketChannel socketChannel) throws Exception {  socketChannel.pipeline().addLast(new FixedLengthFrameDecoder(20));  socketChannel.pipeline().addLast(new StringDecoder());  socketChannel.pipeline().addLast(new EchoServerHandler2());  }  });  // 绑定端口，同步等待成功  ChannelFuture f = b.bind(port).sync();   // 等待服务端监听端口关闭  f.channel().closeFuture().sync();  } finally {  bossGroup.shutdownGracefully();  workerGroup.shutdownGracefully();  }  }   public static void main(String[] args) throws Exception {  new EchoServer2().bind(8080);  }   }  package com.netty.study.chapter05;  import io.netty.buffer.ByteBuf; import io.netty.buffer.Unpooled; import io.netty.channel.ChannelHandlerAdapter; import io.netty.channel.ChannelHandlerContext;  public class EchoServerHandler2 extends ChannelHandlerAdapter {   int counter = 0;   @Override  public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {  System.*out*.println("Receive client : [" + msg + "]");  }   @Override  public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws Exception {  cause.printStackTrace();  ctx.close();// 发生异常，关闭链路  } } |

# 编码技术

# MessagePack

# Protobuf

# JBoss

# Http协议开发应用

## 10.1 Http协议介绍

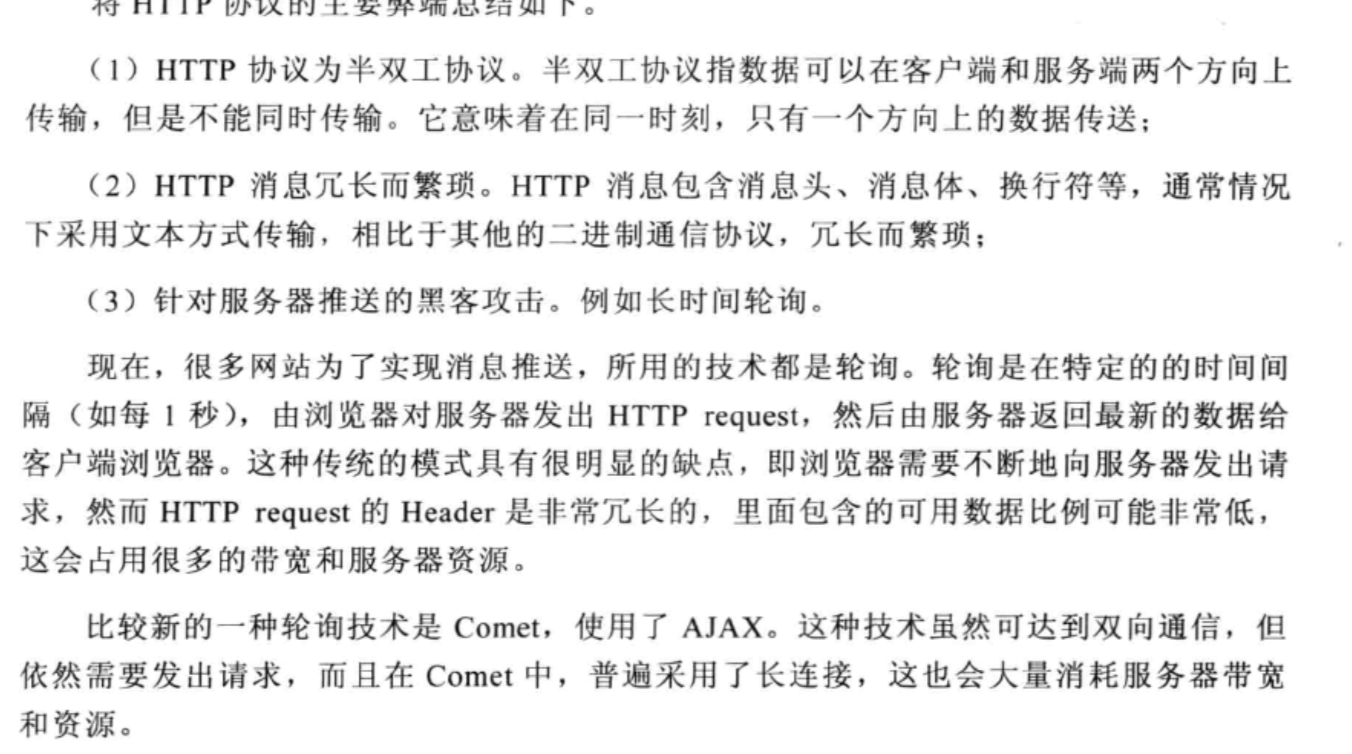
## 10.2 Netty Http服务器

文件服务器

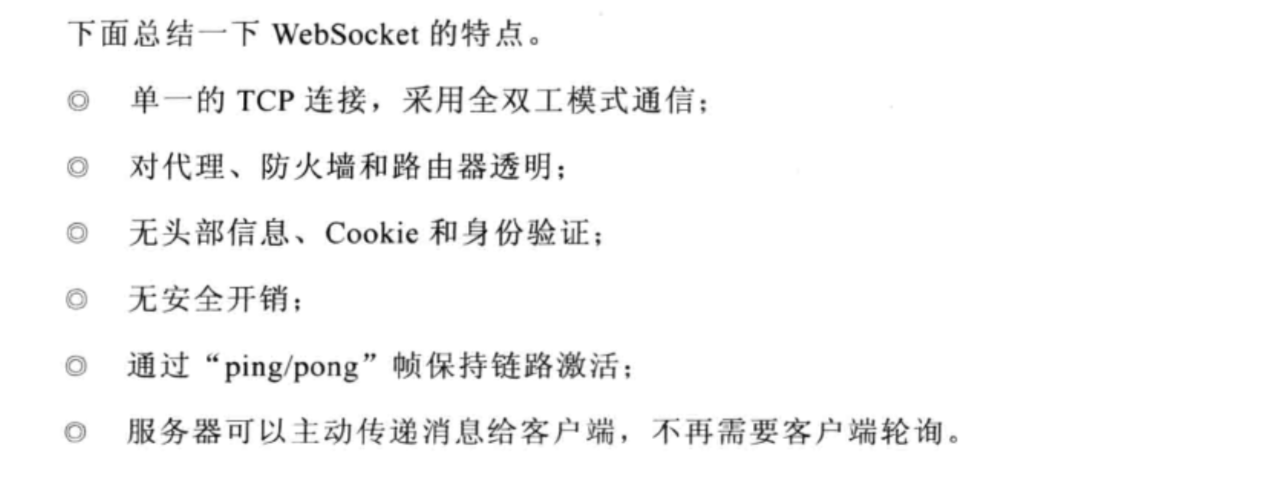
|  |
| --- |
| package com.netty.study.chapter10;  import io.netty.bootstrap.ServerBootstrap; import io.netty.channel.ChannelFuture; import io.netty.channel.ChannelInitializer; import io.netty.channel.EventLoopGroup; import io.netty.channel.nio.NioEventLoopGroup; import io.netty.channel.socket.SocketChannel; import io.netty.channel.socket.nio.NioServerSocketChannel; import io.netty.handler.codec.http.HttpObjectAggregator; import io.netty.handler.codec.http.HttpRequestDecoder; import io.netty.handler.codec.http.HttpResponseEncoder; import io.netty.handler.stream.ChunkedWriteHandler;  public class HttpFileServer {   private static final String *DEFAULT\_URL* = "/src/main/java/com/netty/study/";   public void run(final int port, final String url) throws Exception {  EventLoopGroup bossGroup = new NioEventLoopGroup();  EventLoopGroup workerGroup = new NioEventLoopGroup();   try {  ServerBootstrap b = new ServerBootstrap();  b.group(bossGroup, workerGroup)  .channel(NioServerSocketChannel.class)  .childHandler(new ChannelInitializer<SocketChannel>() {  @Override  protected void initChannel(SocketChannel ch) throws Exception {  ch.pipeline().addLast("http-decoder",  new HttpRequestDecoder()); // 请求消息解码器  ch.pipeline().addLast("http-aggregator",  new HttpObjectAggregator(65536));// 目的是将多个消息转换为单一的request或者response对象  ch.pipeline().addLast("http-encoder",  new HttpResponseEncoder());//响应解码器  ch.pipeline().addLast("http-chunked",  new ChunkedWriteHandler());//目的是支持异步大文件传输（）  ch.pipeline().addLast("fileServerHandler",  new HttpFileServerHandler(url));// 业务逻辑  }  });  ChannelFuture future = b.bind("150.223.208.248", port).sync();  System.*out*.println("Http 文件目录服务器启动，网址是： http://150.223.208.248:" + port + url);   future.channel().closeFuture().sync();  } finally {  bossGroup.shutdownGracefully();  workerGroup.shutdownGracefully();  }  }   public static void main(String[] args) throws Exception {  int port = 8080;   new HttpFileServer().run(port, *DEFAULT\_URL*);    }  }  package com.netty.study.chapter10;  import io.netty.buffer.ByteBuf; import io.netty.buffer.Unpooled; import io.netty.channel.\*; import io.netty.handler.codec.http.\*; import io.netty.handler.stream.ChunkedFile; import io.netty.util.CharsetUtil;  import javax.activation.MimetypesFileTypeMap; import java.io.File; import java.io.FileNotFoundException; import java.io.RandomAccessFile; import java.io.UnsupportedEncodingException; import java.net.URLDecoder; import java.util.regex.Pattern;  import static io.netty.handler.codec.http.HttpVersion.*HTTP\_1\_1*;  public class HttpFileServerHandler extends SimpleChannelInboundHandler<FullHttpRequest> {    private String url;  private static final Pattern *INSECURE\_URI* = Pattern.*compile*(".\*[<>&\"].\*");   private static final Pattern *ALLOWED\_FILE\_NAME* = Pattern  .*compile*("[A-Za-z0-9][-\_A-Za-z0-9\\.]\*");   public HttpFileServerHandler(String url) {  this.url = url;  }   @Override  protected void messageReceived(ChannelHandlerContext ctx, FullHttpRequest fullHttpRequest) throws Exception {  if (!fullHttpRequest.getDecoderResult().isSuccess()) {  sendError(ctx, HttpResponseStatus.*BAD\_REQUEST*);  return;  }   if (fullHttpRequest.getMethod() != HttpMethod.*GET*) {  sendError(ctx, HttpResponseStatus.*METHOD\_NOT\_ALLOWED*);  return;  }   final String uri = fullHttpRequest.getUri();  final String path = sanitizeUri(uri);    if (path == null) {  sendError(ctx, HttpResponseStatus.*FORBIDDEN*);  return;  }   File file = new File(path);  if (file.isHidden() || !file.exists()) {  sendError(ctx, HttpResponseStatus.*NOT\_FOUND*);  }   if (file.isDirectory()) {  if (uri.endsWith("/")) {  sendListing(ctx, file);  } else {  sendRedirect(ctx, uri + '/');  }  return;  }  if (!file.isFile()) {  sendError(ctx, HttpResponseStatus.*FORBIDDEN*);  return;  }    RandomAccessFile randomAccessFile = null;  try {  randomAccessFile = new RandomAccessFile(file, "r");// 以只读的方式打开文件  } catch (FileNotFoundException e) {  sendError(ctx, HttpResponseStatus.*NOT\_FOUND*);  return;  }   long fileLength = randomAccessFile.length();   DefaultHttpResponse response = new DefaultHttpResponse(*HTTP\_1\_1*, HttpResponseStatus.*OK*);  HttpHeaders.*setContentLength*(response, fileLength);  setContentTypeHeader(response, file);    if (HttpHeaders.*isKeepAlive*(fullHttpRequest)) {  response.headers().set(HttpHeaders.Names.*CONNECTION*, HttpHeaders.Values.*KEEP\_ALIVE*);  }   ctx.write(response);  ChannelFuture sendFileFuture;  sendFileFuture = ctx.write(new ChunkedFile(randomAccessFile, 0, fileLength, 8192), ctx.newProgressivePromise());   sendFileFuture.addListener(new ChannelProgressiveFutureListener() {  @Override  public void operationProgressed(ChannelProgressiveFuture future, long progress, long total) throws Exception {  if (total < 0) {  System.*out*.println("Transfer progress : " + progress);  } else {  System.*out*.println("Transfer progerss : " + progress + "/" + total);  }  }   @Override  public void operationComplete(ChannelProgressiveFuture future) throws Exception {  System.*out*.println("Transfer complete.");  }  });    ChannelFuture lastContentFuture = ctx.writeAndFlush(LastHttpContent.*EMPTY\_LAST\_CONTENT*);  if (!HttpHeaders.*isKeepAlive*(fullHttpRequest)) {  lastContentFuture.addListener(ChannelFutureListener.*CLOSE*);  }   }   private void sendRedirect(ChannelHandlerContext ctx, String newUri) {  FullHttpResponse response = new DefaultFullHttpResponse(*HTTP\_1\_1*, HttpResponseStatus.*FOUND*);  response.headers().set(HttpHeaders.Names.*LOCATION*, newUri);  ctx.writeAndFlush(response).addListener(ChannelFutureListener.*CLOSE*);  }   private void sendListing(ChannelHandlerContext ctx, File dir) {  FullHttpResponse response = new DefaultFullHttpResponse(*HTTP\_1\_1*, HttpResponseStatus.*OK*);  response.headers().set(HttpHeaders.Names.*CONTENT\_TYPE*, "text/html; charset=UTF-8");  StringBuilder buf = new StringBuilder();  String dirPath = dir.getPath();  buf.append("<!DOCTYPE html>\r\n");  buf.append("<html><head><title>");  buf.append(dirPath);  buf.append(" 目录：");  buf.append("</title></head><body>\r\n");  buf.append("<h3>");  buf.append(dirPath).append(" 目录：");  buf.append("</h3>\r\n");  buf.append("<ul>");  buf.append("<li>链接：<a href=\"../\">..</a></li>\r\n");  for (File f : dir.listFiles()) {  if (f.isHidden() || !f.canRead()) {  continue;  }  String name = f.getName();  if (!*ALLOWED\_FILE\_NAME*.matcher(name).matches()) {  continue;  }  buf.append("<li>链接：<a href=\"");  buf.append(name);  buf.append("\">");  buf.append(name);  buf.append("</a></li>\r\n");  }  buf.append("</ul></body></html>\r\n");  ByteBuf buffer = Unpooled.*copiedBuffer*(buf, CharsetUtil.*UTF\_8*);  response.content().writeBytes(buffer);  buffer.release();  ctx.writeAndFlush(response).addListener(ChannelFutureListener.*CLOSE*);  }   @Override  public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws Exception {  cause.printStackTrace();  if (ctx.channel().isActive()) {  sendError(ctx, HttpResponseStatus.*INTERNAL\_SERVER\_ERROR*);  }  }   private String sanitizeUri(String uri) {  try {  URLDecoder.*decode*(uri, "UTF-8");  } catch (UnsupportedEncodingException e) {  e.printStackTrace();  try {  uri = URLDecoder.*decode*(uri, "ISO-8859-1");  } catch (UnsupportedEncodingException ex) {  ex.printStackTrace();  throw new Error();  }  }   if (!uri.startsWith(url)) {  return null;  }   if (!uri.startsWith("/")) {  return null;  }   uri = uri.replace('/', File.*separatorChar*);   if (uri.contains(File.*separator* + ".") || uri.contains('.' + File.*separator*) || uri.startsWith(".") ||  uri.endsWith(".") || *INSECURE\_URI*.matcher(uri).matches()) {  return null;  }   return System.*getProperty*("user.dir") + File.*separator* + uri;  }    private void setContentTypeHeader(DefaultHttpResponse response, File file) {  MimetypesFileTypeMap mimeTypesMap = new MimetypesFileTypeMap();  response.headers().set(HttpHeaders.Names.*CONTENT\_TYPE*,  mimeTypesMap.getContentType(file.getPath()));   }    private void sendError(ChannelHandlerContext ctx, HttpResponseStatus status) {  FullHttpResponse response = new DefaultFullHttpResponse(*HTTP\_1\_1*,  status, Unpooled.*copiedBuffer*("Failure: " + status.toString()  + "\r\n", CharsetUtil.*UTF\_8*));  response.headers().set(HttpHeaders.Names.*CONTENT\_TYPE*, "text/plain; charset=UTF-8");  ctx.writeAndFlush(response).addListener(ChannelFutureListener.*CLOSE*);  } } |

# WebSocket协议开发

## 11.1 HTTP协议的弊端



## 11.2 WebSocket入门



## 11.3 Netty WebSocket协议开发

服务端

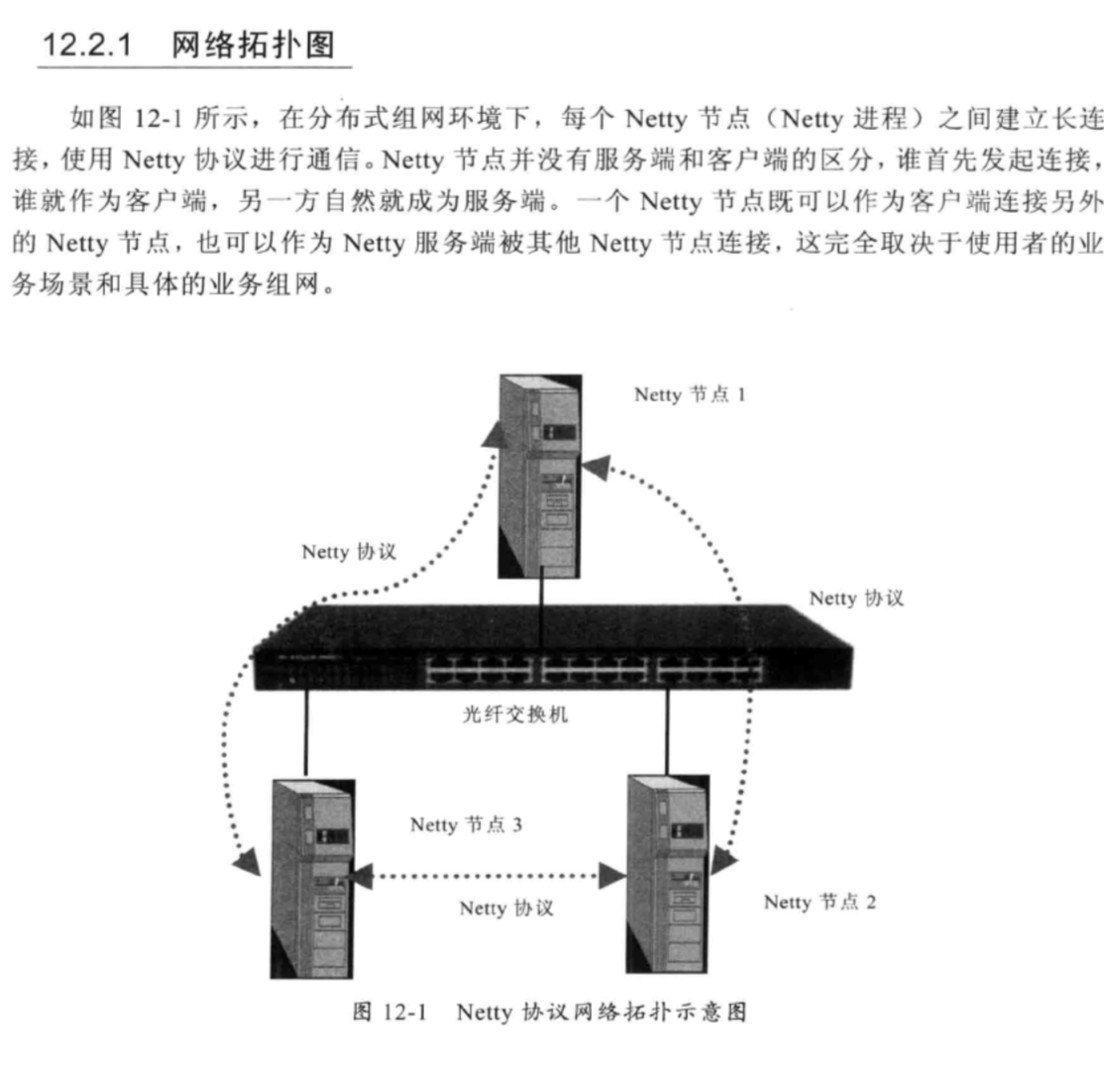
|  |
| --- |
| package com.netty.study.chapter11;  import io.netty.bootstrap.ServerBootstrap; import io.netty.channel.Channel; import io.netty.channel.ChannelInitializer; import io.netty.channel.ChannelPipeline; import io.netty.channel.EventLoopGroup; import io.netty.channel.nio.NioEventLoopGroup; import io.netty.channel.socket.SocketChannel; import io.netty.channel.socket.nio.NioServerSocketChannel; import io.netty.handler.codec.http.HttpObjectAggregator; import io.netty.handler.codec.http.HttpServerCodec; import io.netty.handler.stream.ChunkedWriteHandler;  public class WebSocketServer {    public void run(int port) throws Exception {  EventLoopGroup bossGroup = new NioEventLoopGroup();  EventLoopGroup workerGroup = new NioEventLoopGroup();   try {  ServerBootstrap b = new ServerBootstrap();  b.group(bossGroup, workerGroup)  .channel(NioServerSocketChannel.class)  .childHandler(new ChannelInitializer<SocketChannel>() {  @Override  protected void initChannel(SocketChannel ch) throws Exception {  ChannelPipeline pipeline = ch.pipeline();  pipeline.addLast("http-codec", new HttpServerCodec());  pipeline.addLast("aggregator", new HttpObjectAggregator(65536));  pipeline.addLast("http-chunked", new ChunkedWriteHandler());  pipeline.addLast("handler", new WebSocketServerHandler());  }  });  Channel ch = b.bind(port).sync().channel();  System.*out*.println("Web socket server started at port " + port + ".");  System.*out*.println("Open your browser and navigate to http://localhost: " + port + "/");  ch.closeFuture().sync();  } finally {  bossGroup.shutdownGracefully();  workerGroup.shutdownGracefully();  }    }   public static void main(String[] args) throws Exception {  new WebSocketServer().run(8080);   }   }  package com.netty.study.chapter11;  import io.netty.buffer.ByteBuf; import io.netty.buffer.Unpooled; import io.netty.channel.ChannelFuture; import io.netty.channel.ChannelFutureListener; import io.netty.channel.ChannelHandlerContext; import io.netty.channel.SimpleChannelInboundHandler; import io.netty.handler.codec.http.\*; import io.netty.handler.codec.http.websocketx.\*; import io.netty.util.CharsetUtil;  import java.util.Date;  public class WebSocketServerHandler extends SimpleChannelInboundHandler<Object> {   private WebSocketServerHandshaker handshaker;    protected void messageReceived(ChannelHandlerContext ctx, Object msg) throws Exception {  // 传通的HTTP接入  if (msg instanceof FullHttpRequest) {  handleHttpRequest(ctx, (FullHttpRequest) msg);  }  // WebSocket 接入  else if (msg instanceof WebSocketFrame) {  handleWebSocketFrame(ctx, (WebSocketFrame) msg);  }  }    @Override  public void channelReadComplete(ChannelHandlerContext ctx) throws Exception {  ctx.flush();  }   @Override  public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws Exception {  cause.printStackTrace();  ctx.close();  }   private void handleWebSocketFrame(ChannelHandlerContext ctx, WebSocketFrame frame) {  // 判断是否关闭链路的指令  if (frame instanceof CloseWebSocketFrame) {  handshaker.close(ctx.channel(), (CloseWebSocketFrame) frame.retain());  return;  }   // 判断是否是Ping消息  if (frame instanceof PingWebSocketFrame) {  ctx.channel().write(new PongWebSocketFrame(frame.content().retain()));  return;  }   // 本例程仅支持文本消息，不支持二进制消息  if (!(frame instanceof TextWebSocketFrame)) {  throw new UnsupportedOperationException(String.*format*("%s frame types not supported", frame.getClass().getName()));  }   // 返回应答消息  String request = ((TextWebSocketFrame) frame).text();   ctx.channel().write(new TextWebSocketFrame(request + ",欢迎使用Netty WebSocket服务，现在时刻：" + new Date().toString()));   }   private void handleHttpRequest(ChannelHandlerContext ctx, FullHttpRequest req) {  if (!req.getDecoderResult().isSuccess() || (!"websocket".equals(req.headers().get("Upgrade")))) {  sendHttpResponse(ctx, req, new DefaultFullHttpResponse(HttpVersion.*HTTP\_1\_1*, HttpResponseStatus.*BAD\_REQUEST*));  return;  }   // 构造握手响应返回，本机测试  WebSocketServerHandshakerFactory wsFactory = new WebSocketServerHandshakerFactory("ws://localhost:8080/websocket", null, false);  handshaker = wsFactory.newHandshaker(req);   if (handshaker == null) {  WebSocketServerHandshakerFactory.*sendUnsupportedWebSocketVersionResponse*(ctx.channel());  } else {  handshaker.handshake(ctx.channel(), req);  }    }   private void sendHttpResponse(ChannelHandlerContext ctx, FullHttpRequest req, FullHttpResponse res) {  // 返回应答客户端  if (res.getStatus().code() != 200) {  ByteBuf buf = Unpooled.*copiedBuffer*(res.getStatus().toString(), CharsetUtil.*UTF\_8*);  res.content().writeBytes(buf);  buf.release();  HttpHeaders.*setContentLength*(res, res.content().readableBytes());  }   // 如果是非Keep-Alive，关闭连接  ChannelFuture f = ctx.channel().writeAndFlush(res);  if (!HttpHeaders.*isKeepAlive*(req) || res.getStatus().code() != 200) {  f.addListener(ChannelFutureListener.*CLOSE*);  }  } } |

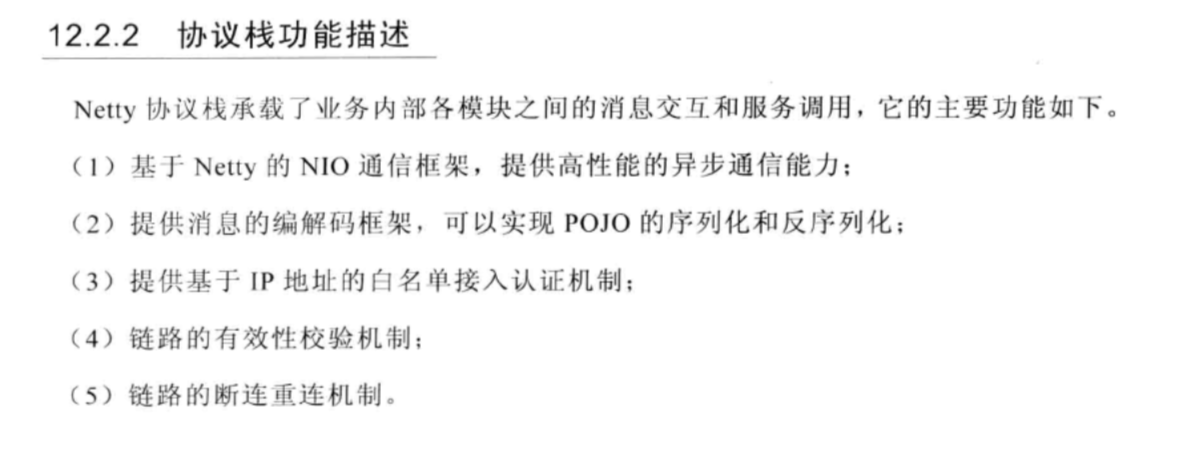
客户端

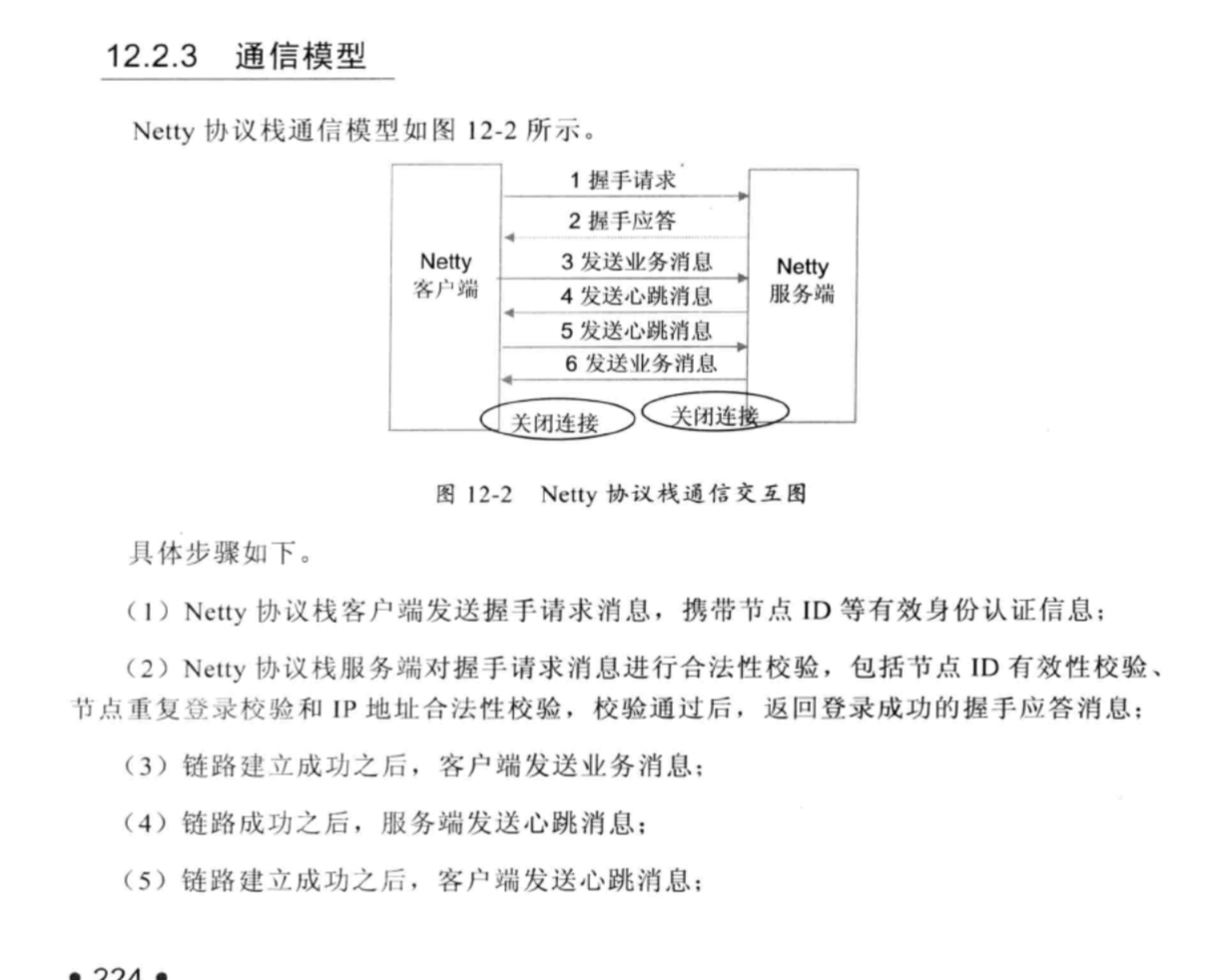
|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <title>时间服务器</title>  </head>  <body>  <script type="text/javascript">    var socket;  if(!window.WebSocket){  window.WebSocket = window.MozWebSocket  }  if(window.WebSocket){  socket = new WebSocket("ws://localhost:8080/websocket");  socket.onmessage = function(event){  var ta = document.getElementById("responseText");  ta.value = "";  ta.value = event.data;  }  socket.onopen = function(event){  var ta = document.getElementById("responseText");  ta.value = "打开WebSocket服务器正常，浏览器支持WebSocket！";  }    socket.onclose = function(event){  var ta = document.getElementById("responseText");  ta.value = "";  ta.value = "WebSocket 关闭";  }  }else{  alert("抱歉，您的浏览器不支持WebSocket协议");  }  function send(message){  if(!window.WebSocket){  return;  }  if(socket.readyState == WebSocket.OPEN){  socket.send(message);  }else{  alert("WebSocket 连接没有建立成功！");  }  }  </script>  <form onsubmit="return false;">  <input type="text" name="message" value="Netty最佳实践">  <br>  <input type="button" value="发送WebSocket 请求消息" onclick="send(this.form.message.value)">  <hr color="blue"/>  <h3>服务端返回的应答消息</h3>  <textarea id="responseText" style="width: 500px;height: 300px;"></textarea>  </form>    </body>  </html> |

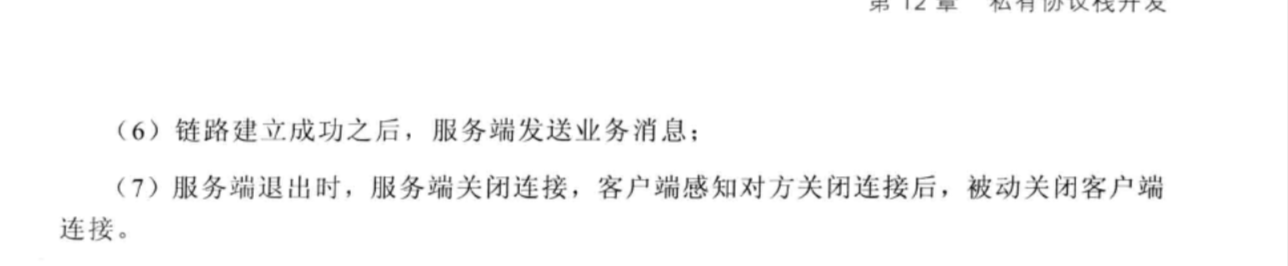
# 私有协议栈开发

## 12.2 Netty协议栈功能的设计



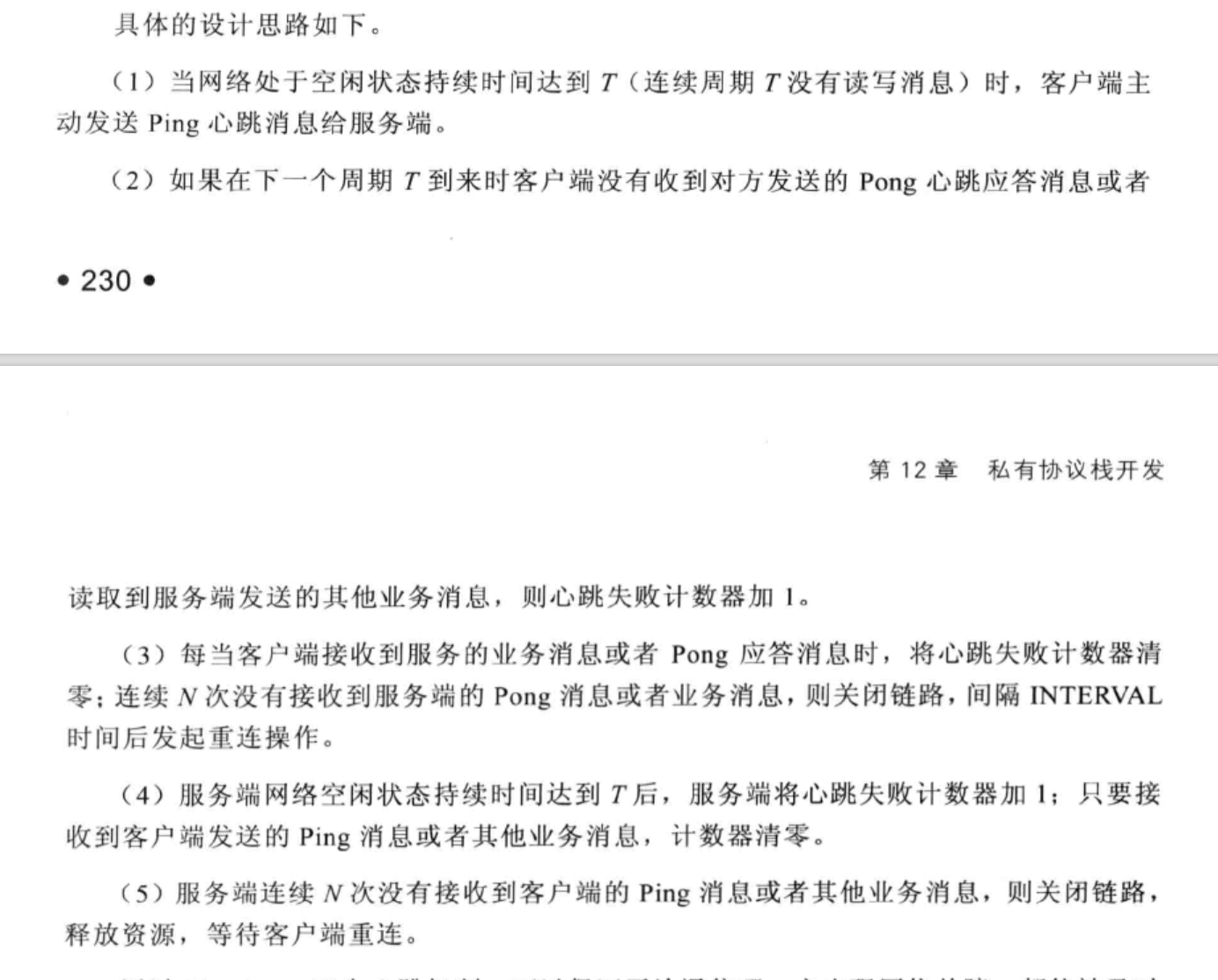






## 12.2.9 可靠性设计

1. 心跳机制



1. 重连机制
2. 重复登录保护
3. 消息缓存重发

## 12.2.10 安全性设计

## 12.2.11 可扩展性设计

## 12.2 Netty协议栈开发